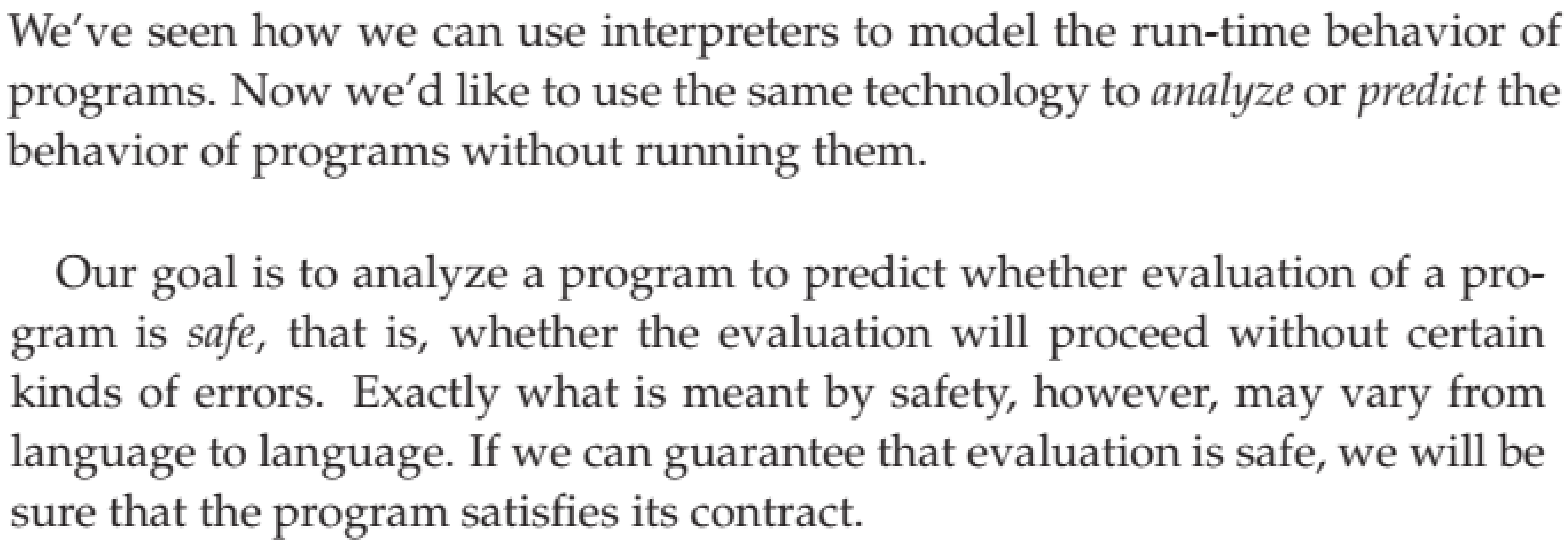
Types

Essentials of Programming Languages (Chapter 7)

BY: MOHAMMAD IZADI

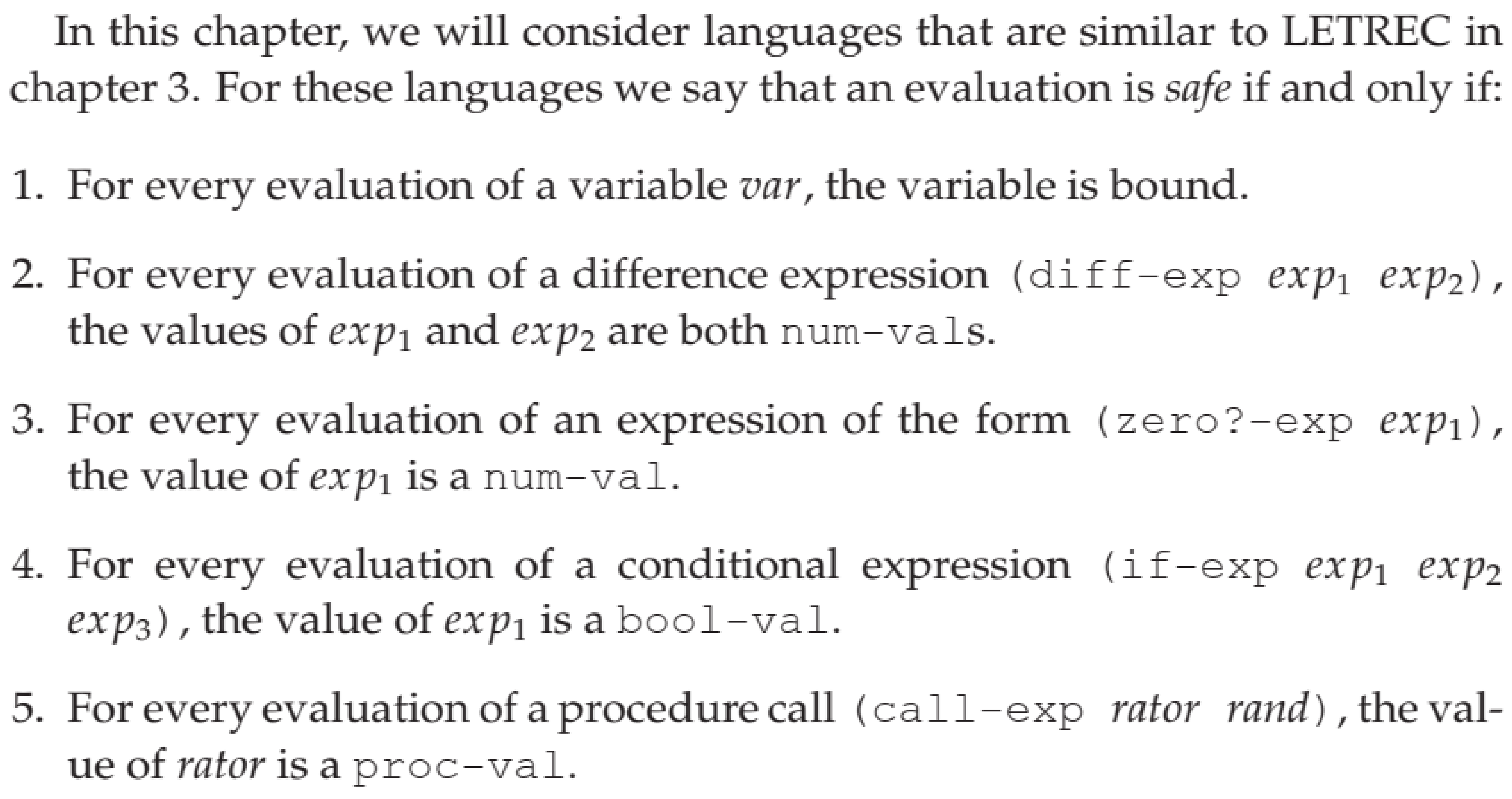
Program Safety Analysis



We've seen how we can use interpreters to model the run-time behavior of programs. Now we'd like to use the same technology to *analyze* or *predict* the behavior of programs without running them.

Our goal is to analyze a program to predict whether evaluation of a program is *safe*, that is, whether the evaluation will proceed without certain kinds of errors. Exactly what is meant by safety, however, may vary from language to language. If we can guarantee that evaluation is safe, we will be sure that the program satisfies its contract.

Safety in LETREC



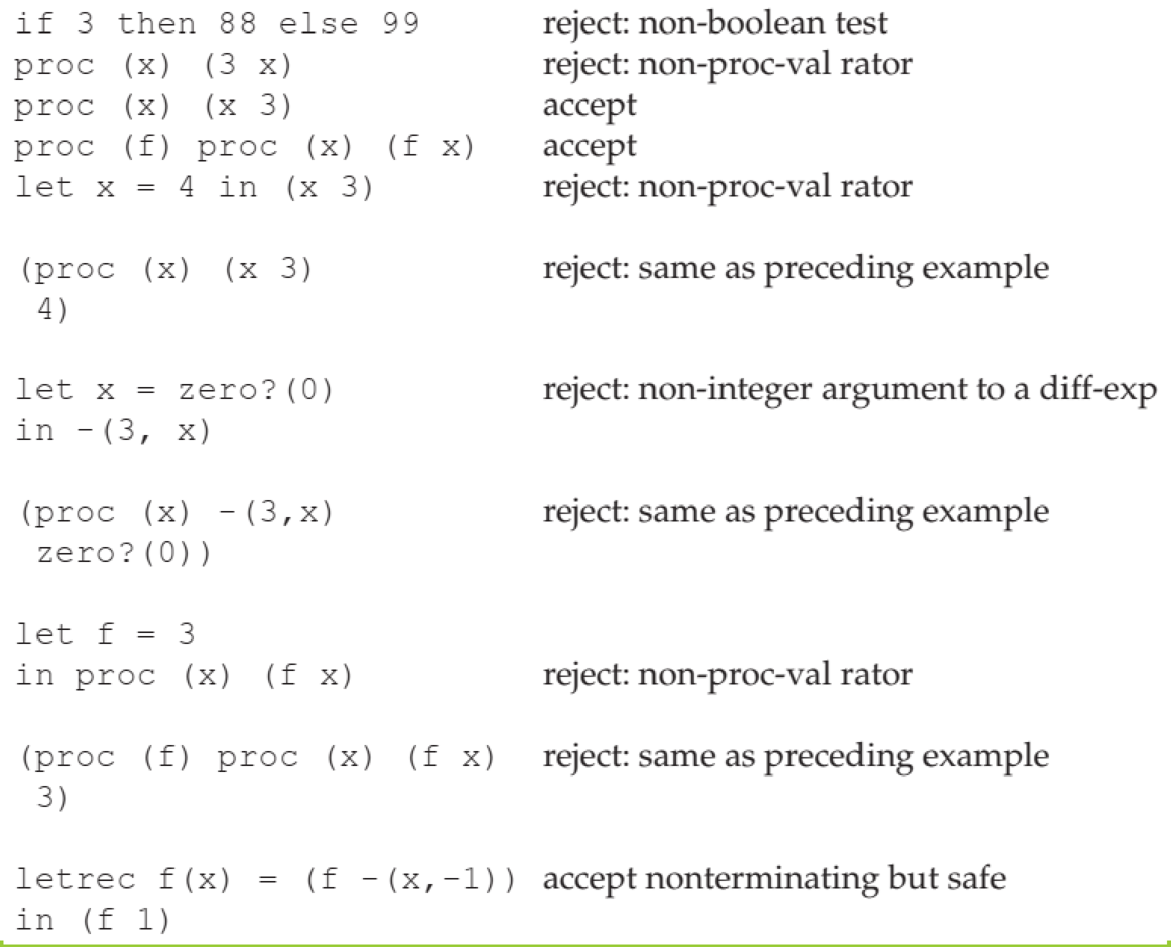
In this chapter, we will consider languages that are similar to LETREC in chapter 3. For these languages we say that an evaluation is *safe* if and only if:

1. For every evaluation of a variable *var*, the variable is bound.
2. For every evaluation of a difference expression `(diff-exp` exp_1 exp_2`)`, the values of exp_1 and exp_2 are both `num-val`s.
3. For every evaluation of an expression of the form `(zero?-exp` exp_1`)`, the value of exp_1 is a `num-val`.
4. For every evaluation of a conditional expression `(if-exp` exp_1 exp_2 exp_3`)`, the value of exp_1 is a `bool-val`.
5. For every evaluation of a procedure call `(call-exp` *rator* *rand*`)`, the value of *rator* is a `proc-val`.

Some Notes

- We call violations of these type safety conditions *type errors*.
- A safe evaluation may still fail for other reasons: division by zero, taking the car of an empty list, etc. We do not include these as part of our definition of safety.
- We do not include non-termination as part of safety because checking for termination is also very difficult (indeed, it is undecidable in general).
- Our goal is to write a procedure that looks at the program text and either accepts or rejects it. Furthermore, we would like our analysis procedure to be conservative: if the analysis accepts the program, then we can be sure evaluation of the program will be safe. If the analysis cannot be sure that evaluation will be safe, it must reject the program. In this case, we say that the analysis is *sound*.

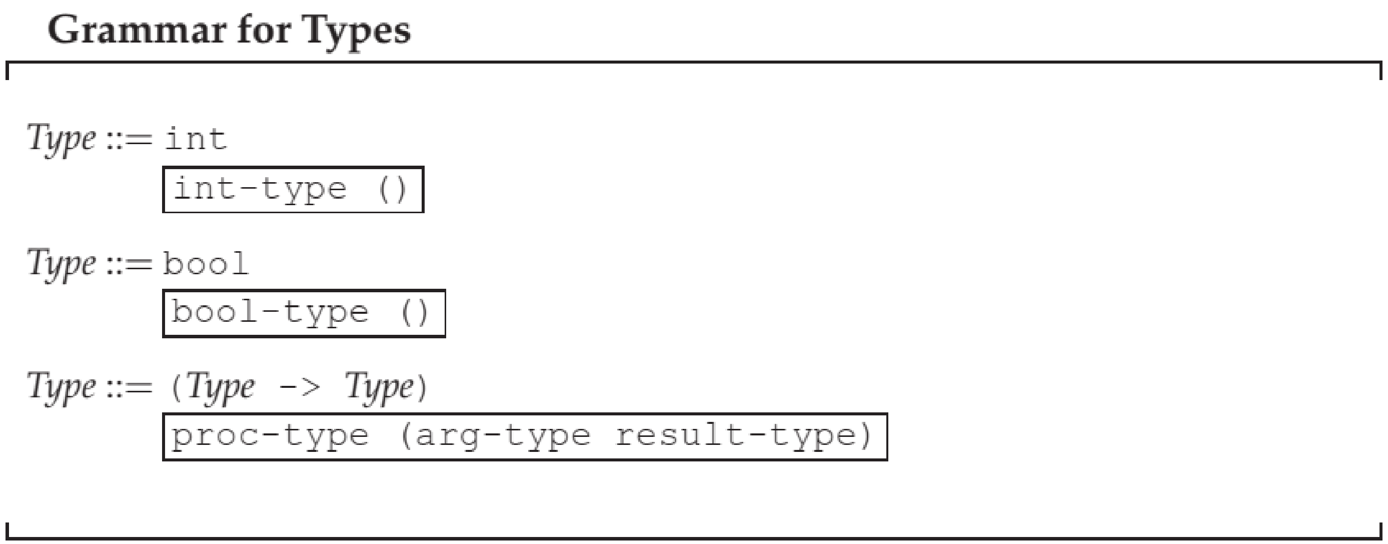
Examples of type analysis



```
if 3 then 88 else 99        reject: non-boolean test
proc (x) (3 x)              reject: non-proc-val rator
proc (x) (x 3)              accept
proc (f) proc (x) (f x)     accept
let x = 4 in (x 3)          reject: non-proc-val rator

(proc (x) (x 3)             reject: same as preceding example
 4)

let x = zero?(0)            reject: non-integer argument to a diff-exp
in -(3, x)

(proc (x) -(3,x)            reject: same as preceding example
 zero?(0))

let f = 3
in proc (x) (f x)           reject: non-proc-val rator

(proc (f) proc (x) (f x)    reject: same as preceding example
 3)

letrec f(x) = (f -(x,-1))   accept nonterminating but safe
in (f 1)
```

Values and Their Types

- Since the safety conditions talk only about *num-val, bool-val*, and *proc-val*, one might think that it would be enough to keep track of these three types. But that is not enough: if all we know is that f is bound to a *proc-val*, then we can not draw any conclusions whatsoever about the value of *(f 1).* We need to keep track of finer information about procedures. This finer information is called the *type structure* of the language.
- Types for LETREC language:



Grammar for Types

```
Type ::= int
         int-type ()

Type ::= bool
         bool-type ()

Type ::= (Type -> Type)
         proc-type (arg-type result-type)
```

Examples of values and their types

- The value of *3* has type *int*.
- The value of *-(33,22)* has type *int*.
- The value of *zero?(11)* has type *bool*.
- The value of *proc (x) -(x,11)* has type *(int -> int)*.
- The value of *proc (x) let y = -(x,11) in -(x,y)* has type *(int -> int)*.
- The value of *proc (x) if x then 11 else 22* has type *(bool -> int)*.
- The value of *proc (x) if x then 11 else zero?(11)* has no type in our type system.

Examples of values and their types

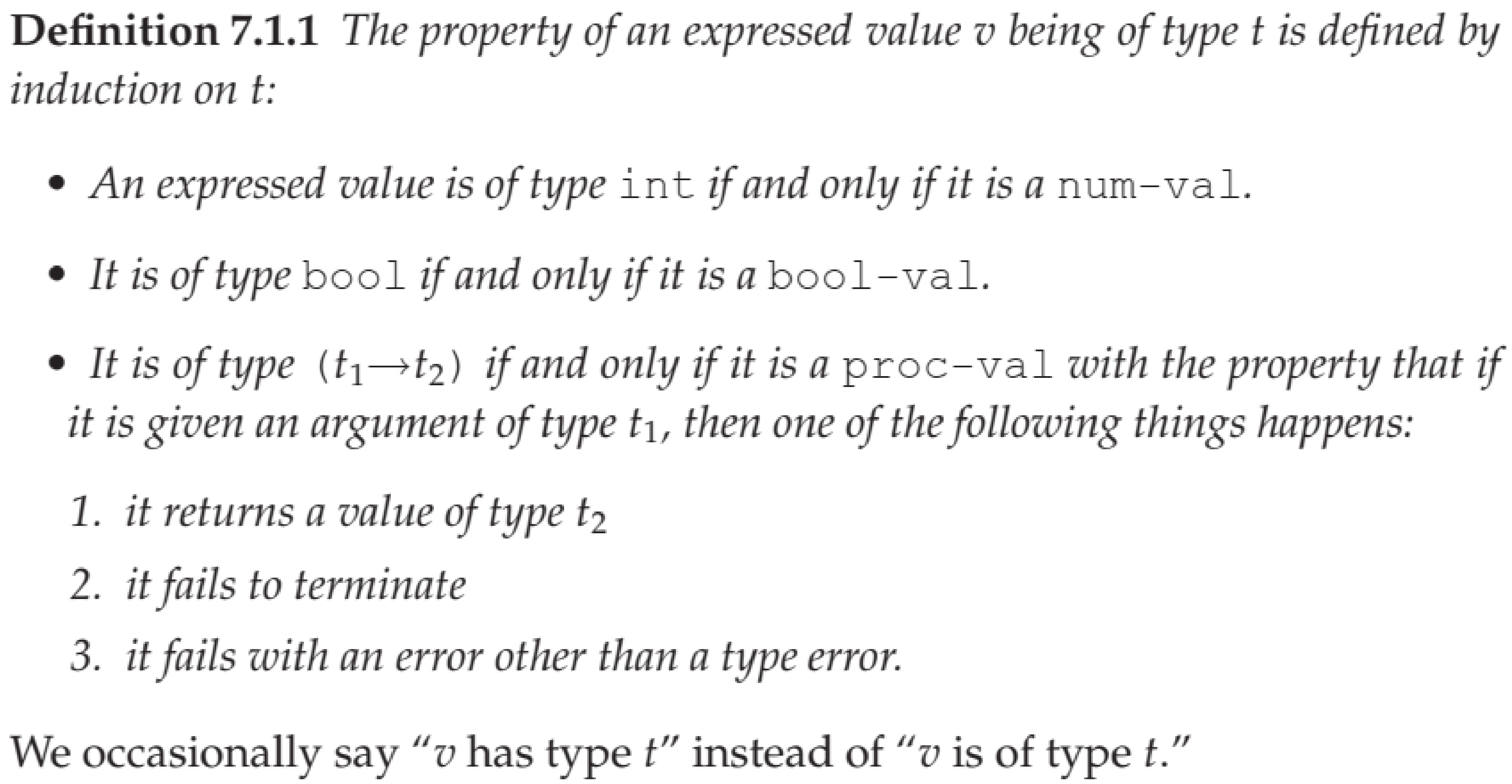
- The value of *proc (x) proc (y) if y then x else 11* has type *(int -> (bool -> int).*

- The value of *proc (f) if (f 3) then 11 else 22* has type ((int -> bool) -> int).

- The value of *proc (f) (f 3)* has type *((int -> t) -> t)* for any type *t*.

- The value of *proc (f) proc (x) (f (f x))* has type *((t -> t) -> (t -> t))* for any type *t*.

Definition of types for values and expressions



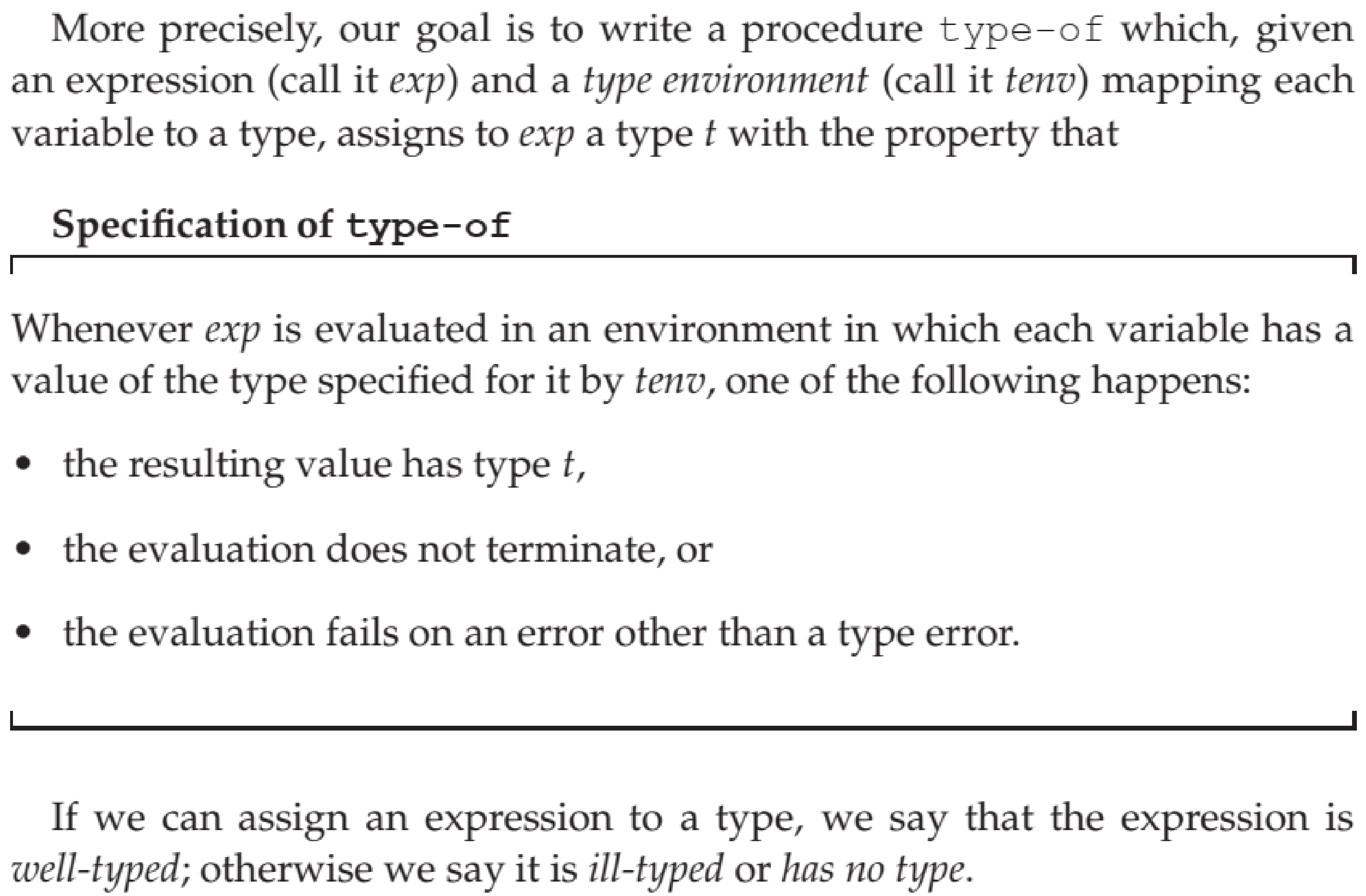
Definition 7.1.1 *The property of an expressed value v being of type t is defined by induction on t:*

- *An expressed value is of type* `int` *if and only if it is a* `num-val`.
- *It is of type* `bool` *if and only if it is a* `bool-val`.
- *It is of type* `(`$t_1 \rightarrow t_2$`)` *if and only if it is a* `proc-val` *with the property that if it is given an argument of type t_1, then one of the following things happens:*

  1. *it returns a value of type t_2*
  2. *it fails to terminate*
  3. *it fails with an error other than a type error.*

We occasionally say "v has type t" instead of "v is of type t."

Assigning a Type to an Expression



More precisely, our goal is to write a procedure `type-of` which, given an expression (call it *exp*) and a *type environment* (call it *tenv*) mapping each variable to a type, assigns to *exp* a type *t* with the property that

Specification of `type-of`

Whenever *exp* is evaluated in an environment in which each variable has a value of the type specified for it by *tenv*, one of the following happens:

- the resulting value has type *t*,
- the evaluation does not terminate, or
- the evaluation fails on an error other than a type error.

If we can assign an expression to a type, we say that the expression is *well-typed*; otherwise we say it is *ill-typed* or *has no type*.

Typing Rules

$$\texttt{(type-of (const-exp } \mathit{num}\texttt{) } \mathit{tenv}\texttt{) = int}$$

$$\texttt{(type-of (var-exp } \mathit{var}\texttt{) } \mathit{tenv}\texttt{) = } \mathit{tenv}\texttt{(}\mathit{var}\texttt{)}$$

$$\frac{\texttt{(type-of } \mathit{exp}_1 \ \mathit{tenv}\texttt{) = int}}{\texttt{(type-of (zero?-exp } \mathit{exp}_1\texttt{) } \mathit{tenv}\texttt{) = bool}}$$

$$\frac{\texttt{(type-of } \mathit{exp}_1 \ \mathit{tenv}\texttt{) = int} \qquad \texttt{(type-of } \mathit{exp}_2 \ \mathit{tenv}\texttt{) = int}}{\texttt{(type-of (diff-exp } \mathit{exp}_1 \ \mathit{exp}_2\texttt{) } \mathit{tenv}\texttt{) = int}}$$

$$\frac{\texttt{(type-of } \mathit{exp}_1 \ \mathit{tenv}\texttt{) = } t_1 \qquad \texttt{(type-of } \mathit{body} \ \texttt{[}\mathit{var}\texttt{=}t_1\texttt{]}\mathit{tenv}\texttt{) = } t_2}{\texttt{(type-of (let-exp } \mathit{var} \ \mathit{exp}_1 \ \mathit{body}\texttt{) } \mathit{tenv}\texttt{) = } t_2}$$

$$\frac{\begin{array}{c}\texttt{(type-of } \mathit{exp}_1 \ \mathit{tenv}\texttt{) = bool} \\ \texttt{(type-of } \mathit{exp}_2 \ \mathit{tenv}\texttt{) = } t \\ \texttt{(type-of } \mathit{exp}_3 \ \mathit{tenv}\texttt{) = } t\end{array}}{\texttt{(type-of (if-exp } \mathit{exp}_1 \ \mathit{exp}_2 \ \mathit{exp}_3\texttt{) } \mathit{tenv}\texttt{) = } t}$$

$$\frac{\texttt{(type-of } \mathit{rator} \ \mathit{tenv}\texttt{) = (}t_1 \rightarrow t_2\texttt{)} \qquad \texttt{(type-of } \mathit{rand} \ \mathit{tenv}\texttt{) = } t_1}{\texttt{(type-of (call-exp } \mathit{rator} \ \mathit{rand}\texttt{) } \mathit{tenv}\texttt{) = } t_2}$$

$$\frac{\texttt{(type-of } \mathit{body} \ \texttt{[}\mathit{var}\texttt{=}t_1\texttt{]}\mathit{tenv}\texttt{) = } t_2}{\texttt{(type-of (proc-exp } \mathit{var} \ \mathit{body}\texttt{) } \mathit{tenv}\texttt{) = (}t_1 \rightarrow t_2\texttt{)}}$$

Tow main type system design streams

- *Type Checking*: In this approach the programmer is required to supply the missing information about the types of bound variables, and the type checker deduces the types of the other expressions and checks them for consistency.

- *Type Inference*: In this approach the type-checker attempts to *infer* the types of the bound variables based on how the variables are used in the program. If the language is carefully designed, the type-checker can infer most or all of these types.

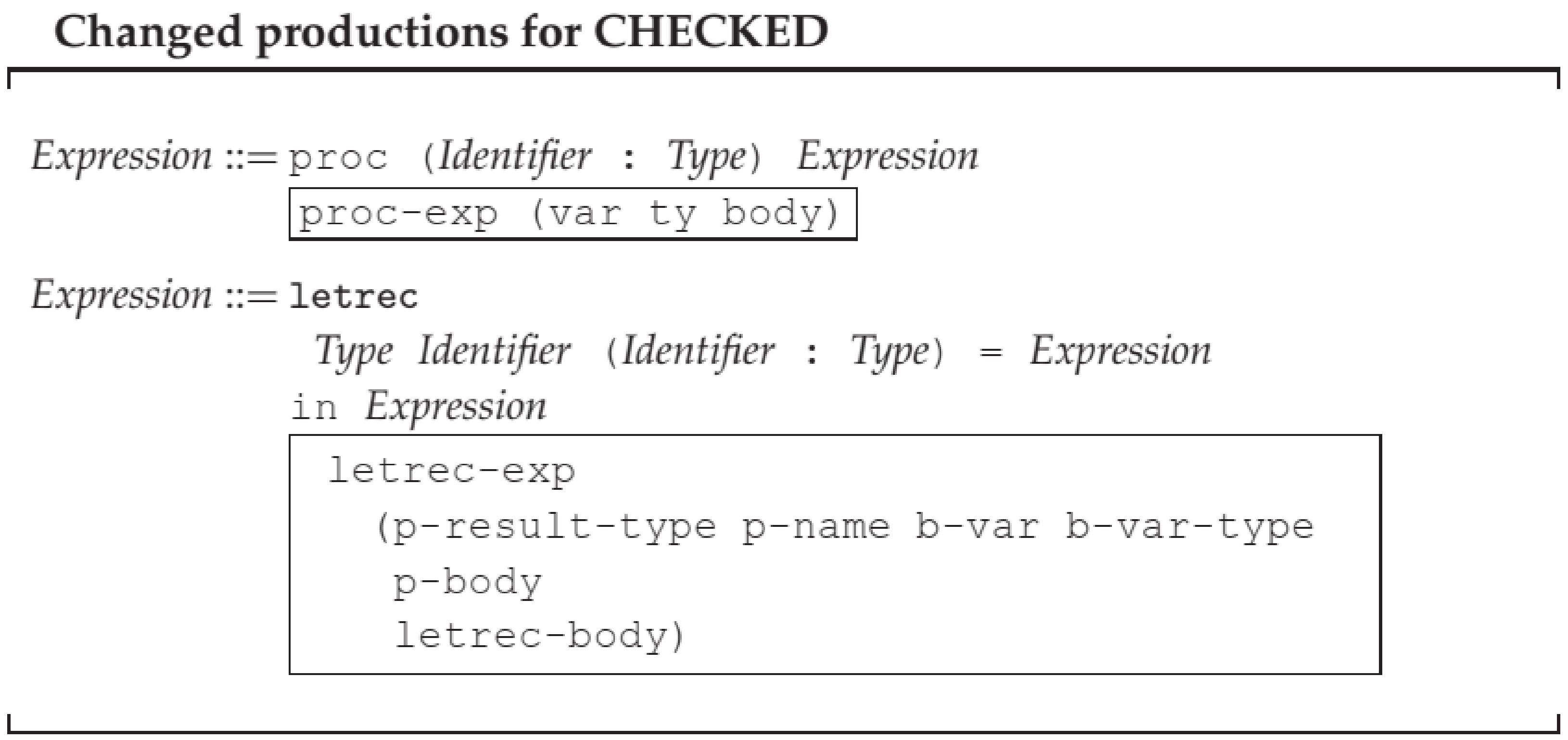
CHECKED: A Type-Checked Language

CHECKED is the same as LETREC, except that we require the programmer to include the types of all bound variables. Here are some example programs in CHECKED.

```
proc (x : int) -(x,1)

letrec
 int double (x : int) = if zero?(x)
                        then 0
                        else -((double -(x,1)), -2)
in double

proc (f : (bool -> int)) proc (n : int) (f zero?(n))
```

The result type of `double` is `int`, but the type of `double` itself is `(int -> int)`, since it is a procedure that takes an integer and returns an integer.

CHECKED: Syntax



Changed productions for CHECKED

Expression ::= `proc` (*Identifier* `:` *Type*) *Expression*

```
proc-exp (var ty body)
```

Expression ::= **`letrec`**

Type Identifier (*Identifier* `:` *Type*) `=` *Expression*

`in` *Expression*

```
letrec-exp
  (p-result-type p-name b-var b-var-type
   p-body
   letrec-body)
```

CHECKED: Type Rules

(in addition to rules in slide 11)

$$\frac{(\texttt{type-of}\ \mathit{body}\ [\mathit{var}{=}t_{\mathit{var}}]\mathit{tenv}) = t_{\mathit{res}}}{(\texttt{type-of}\ (\texttt{proc-exp}\ \mathit{var}\ t_{\mathit{var}}\ \mathit{body})\ \mathit{tenv}) = (t_{\mathit{var}} \longrightarrow t_{\mathit{res}})}$$

$$\frac{\begin{array}{c}(\texttt{type-of}\ e_{\mathit{proc\text{-}body}}\ [\mathit{var}{=}t_{\mathit{var}}][p = (t_{\mathit{var}} \longrightarrow t_{\mathit{res}})]\mathit{tenv}) = t_{\mathit{res}} \\ (\texttt{type-of}\ e_{\mathit{letrec\text{-}body}}\ [p = (t_{\mathit{var}} \longrightarrow t_{\mathit{res}})]\mathit{tenv}) = t\end{array}}{(\texttt{type-of}\ \texttt{letrec}\ t_{\mathit{res}}\ p\ (\mathit{var} : t_{\mathit{var}}) = e_{\mathit{proc\text{-}body}}\ \texttt{in}\ e_{\mathit{letrec\text{-}body}}\ \mathit{tenv}) = t}$$

CHECKED: Type Checker

check-equal-type! : *Type* × *Type* × *Exp* → *Unspecified*

```
(define check-equal-type!
  (lambda (ty1 ty2 exp)
    (if (not (equal? ty1 ty2))
      (report-unequal-types ty1 ty2 exp))))
```

report-unequal-types : *Type* × *Type* × *Exp* → *Unspecified*

```
(define report-unequal-types
  (lambda (ty1 ty2 exp)
    (eopl:error 'check-equal-type!
      "Types didn't match: ~s != ~a in~%~a"
      (type-to-external-form ty1)
      (type-to-external-form ty2)
      exp)))
```

type-to-external-form : *Type* → *List*

```
(define type-to-external-form
  (lambda (ty)
    (cases type ty
      (int-type () 'int)
      (bool-type () 'bool)
      (proc-type (arg-type result-type)
        (list
          (type-to-external-form arg-type)
          '->
          (type-to-external-form result-type))))))
```

CHECKED: Type Checker (cont'd)

Tenv = *Var* → *Type*

type-of-program : *Program* → *Type*

```
(define type-of-program
  (lambda (pgm)
    (cases program pgm
      (a-program (exp1) (type-of exp1 (init-tenv))))))
```

type-of : *Exp* × *Tenv* → *Type*

```
(define type-of
  (lambda (exp tenv)
    (cases expression exp
```

$$\boxed{(\texttt{type-of}\ num\ tenv) = \texttt{int}}$$

```
      (const-exp (num) (int-type))
```

$$\boxed{(\texttt{type-of}\ var\ tenv) = tenv(var)}$$

```
      (var-exp (var) (apply-tenv tenv var))
```

$$\boxed{\frac{(\texttt{type-of}\ e_1\ tenv) = \texttt{int} \qquad (\texttt{type-of}\ e_2\ tenv) = \texttt{int}}{(\texttt{type-of}\ (\texttt{diff-exp}\ e_1\ e_2)\ tenv) = \texttt{int}}}$$

```
      (diff-exp (exp1 exp2)
        (let ((ty1 (type-of exp1 tenv))
              (ty2 (type-of exp2 tenv)))
          (check-equal-type! ty1 (int-type) exp1)
          (check-equal-type! ty2 (int-type) exp2)
          (int-type)))
```

$$\boxed{\frac{(\texttt{type-of}\ e_1\ tenv) = \texttt{int}}{(\texttt{type-of}\ (\texttt{zero?-exp}\ e_1)\ tenv) = \texttt{bool}}}$$

```
(zero?-exp (exp1)
  (let ((ty1 (type-of exp1 tenv)))
    (check-equal-type! ty1 (int-type) exp1)
    (bool-type)))
```

$$\boxed{\frac{\begin{array}{c}(\texttt{type-of}\ e_1\ tenv) = \texttt{bool}\\ (\texttt{type-of}\ e_2\ tenv) = t\\ (\texttt{type-of}\ e_3\ tenv) = t\end{array}}{(\texttt{type-of}\ (\texttt{if-exp}\ e_1\ e_2\ e_3)\ tenv) = t}}$$

```
(if-exp (exp1 exp2 exp3)
  (let ((ty1 (type-of exp1 tenv))
        (ty2 (type-of exp2 tenv))
        (ty3 (type-of exp3 tenv)))
    (check-equal-type! ty1 (bool-type) exp1)
    (check-equal-type! ty2 ty3 exp)
    ty2))
```

CHECKED: Type Checker (cont'd)

$$\boxed{\frac{(\texttt{type-of}\ e_1\ tenv) = t_1 \qquad (\texttt{type-of}\ body\ [var{=}t_1]tenv) = t_2}{(\texttt{type-of}\ (\texttt{let-exp}\ var\ e_1\ body)\ tenv) = t_2}}$$

```
(let-exp (var exp1 body)
  (let ((exp1-type (type-of exp1 tenv)))
    (type-of body
      (extend-tenv var exp1-type tenv))))
```

$$\boxed{\frac{(\texttt{type-of}\ body\ [var{=}t_{var}]tenv) = t_{res}}{(\texttt{type-of}\ (\texttt{proc-exp}\ var\ t_{var}\ body)\ tenv) = (t_{var} \to t_{res})}}$$

```
(proc-exp (var var-type body)
  (let ((result-type
          (type-of body
            (extend-tenv var var-type tenv))))
    (proc-type var-type result-type)))
```

$$\boxed{\frac{(\texttt{type-of}\ rator\ tenv) = (t_1 \to t_2) \qquad (\texttt{type-of}\ rand\ tenv) = t_1}{(\texttt{type-of}\ (\texttt{call-exp}\ rator\ rand)\ tenv) = t_2}}$$

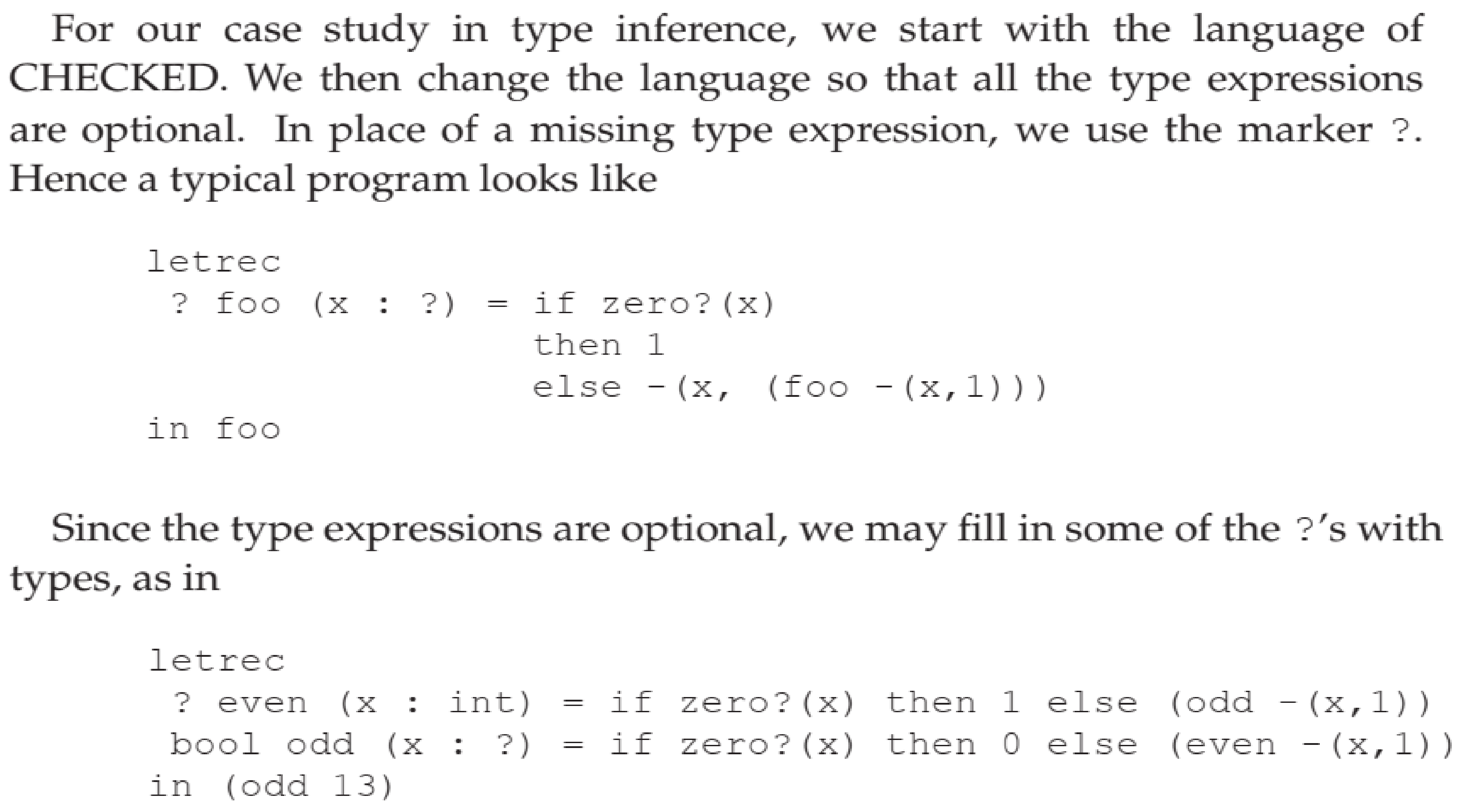
```
(call-exp (rator rand)
  (let ((rator-type (type-of rator tenv))
        (rand-type  (type-of rand tenv)))
    (cases type rator-type
      (proc-type (arg-type result-type)
        (begin
          (check-equal-type! arg-type rand-type rand)
          result-type))
      (else
        (report-rator-not-a-proc-type
          rator-type rator)))))
```

CHECKED: Type Checker (cont’d)

$$\boxed{\frac{\begin{array}{c}(\texttt{type-of}\ e_{proc\text{-}body}\ \ [var{=}t_{var}]\,[p=(t_{var}\ \to\ t_{res})]\,tenv)\ =\ t_{res}\\ (\texttt{type-of}\ e_{letrec\text{-}body}\ \ [p=(t_{var}\ \to\ t_{res})]\,tenv)\ =\ t\end{array}}{(\texttt{type-of letrec}\ t_{res}\ p\ (var : t_{var}) = e_{proc\text{-}body}\ \texttt{in}\ e_{letrec\text{-}body}\ \ tenv)\ =\ t}}$$

```
(letrec-exp (p-result-type p-name b-var b-var-type
              p-body letrec-body)
  (let ((tenv-for-letrec-body
          (extend-tenv p-name
            (proc-type b-var-type p-result-type)
            tenv)))
    (let ((p-body-type
            (type-of p-body
              (extend-tenv b-var b-var-type
                tenv-for-letrec-body))))
      (check-equal-type!
        p-body-type p-result-type p-body)
      (type-of letrec-body tenv-for-letrec-body))))))
```

INFERRED: A Language with Type Inference



For our case study in type inference, we start with the language of CHECKED. We then change the language so that all the type expressions are optional. In place of a missing type expression, we use the marker `?`. Hence a typical program looks like

```
letrec
 ? foo (x : ?) = if zero?(x)
                 then 1
                 else -(x, (foo -(x,1)))
in foo
```

Since the type expressions are optional, we may fill in some of the `?`'s with types, as in

```
letrec
 ? even (x : int) = if zero?(x) then 1 else (odd -(x,1))
 bool odd (x : ?) = if zero?(x) then 0 else (even -(x,1))
in (odd 13)
```

INFERRED: Syntax

Optional-type ::= `?`

```
no-type ()
```

Optional-type ::= *Type*

```
a-type (ty)
```

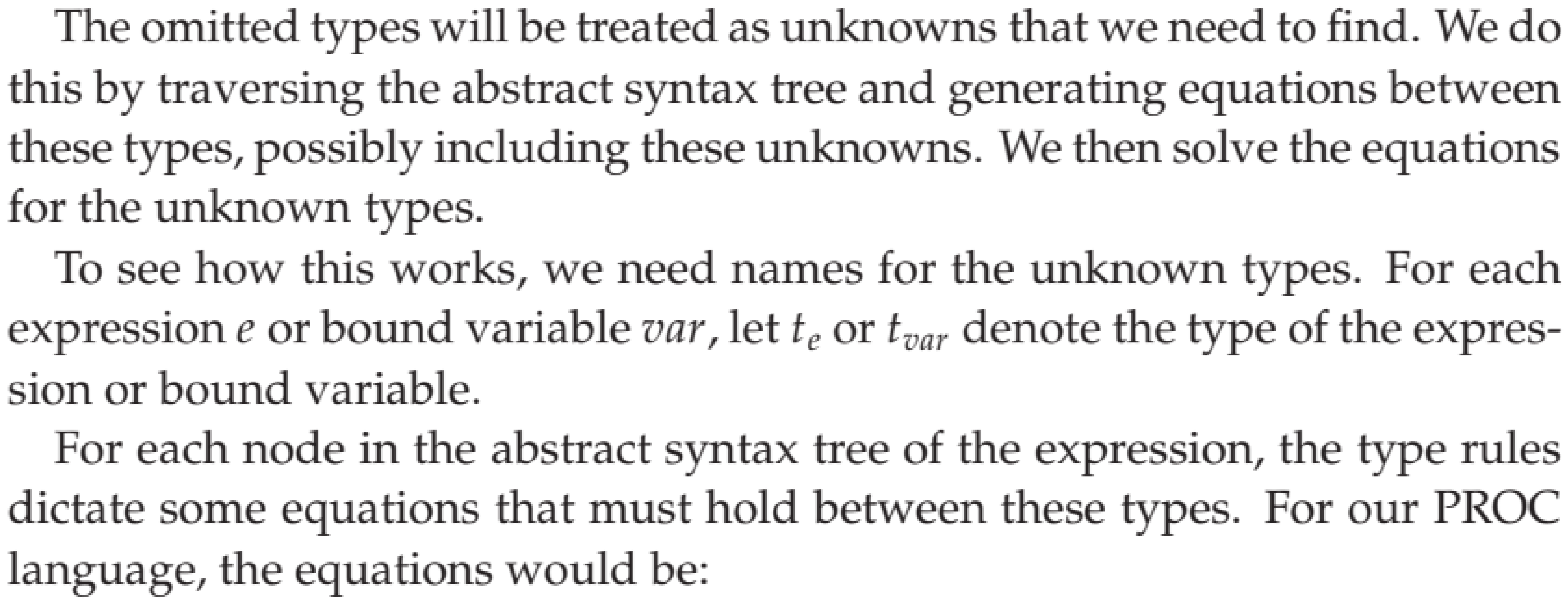
Expression ::= `proc` (*Identifier* `:` *Optional-type*) *Expression*

```
proc-exp (var otype body)
```

Expression ::= `letrec`
Optional-type *Identifier* (*Identifier* `:` *Optional-type*) `=` *Expression*
`in` *Expression*

```
letrec-exp
  (p-result-otype p-name
   b-var b-var-otype p-body
   letrec-body)
```

Finding the omitted types over AST



The omitted types will be treated as unknowns that we need to find. We do this by traversing the abstract syntax tree and generating equations between these types, possibly including these unknowns. We then solve the equations for the unknown types.

To see how this works, we need names for the unknown types. For each expression e or bound variable var, let t_e or t_{var} denote the type of the expression or bound variable.

For each node in the abstract syntax tree of the expression, the type rules dictate some equations that must hold between these types. For our PROC language, the equations would be:

Main Type Equations

$$\begin{aligned}
&(\texttt{diff-exp}\ e_1\ e_2) &:\ & t_{e_1} = \texttt{int} \\
& & & t_{e_2} = \texttt{int} \\
& & & t_{(\texttt{diff-exp}\ e_1\ e_2)} = \texttt{int}
\end{aligned}$$

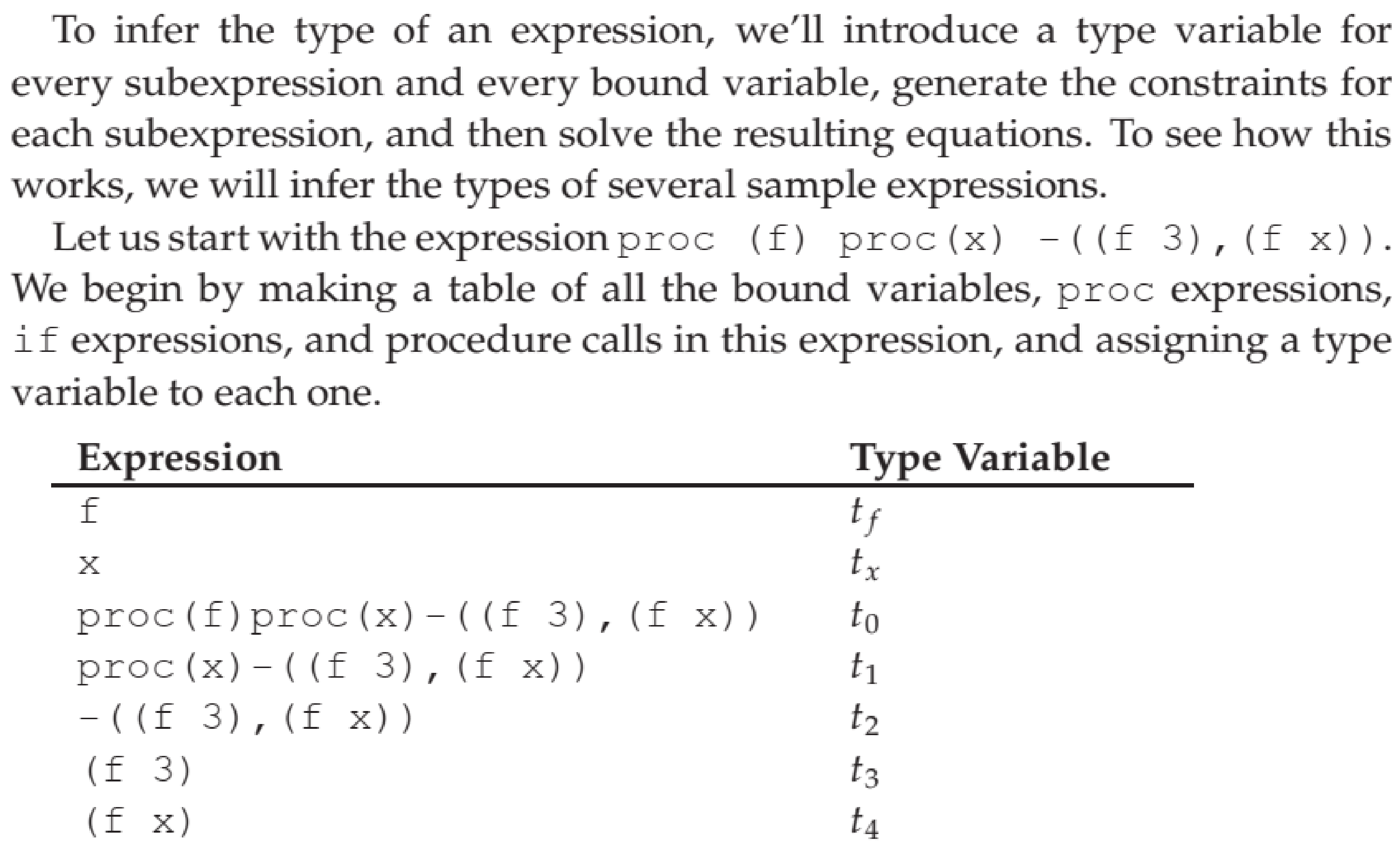
$$\begin{aligned}
&(\texttt{zero?-exp}\ e_1) &:\ & t_{e_1} = \texttt{int} \\
& & & t_{(\texttt{zero?-exp}\ e_1)} = \texttt{bool}
\end{aligned}$$

$$\begin{aligned}
&(\texttt{if-exp}\ e_1\ e_2\ e_3) &:\ & t_{e_1} = \texttt{bool} \\
& & & t_{e_2} = t_{(\texttt{if-exp}\ e_1\ e_2\ e_3)} \\
& & & t_{e_3} = t_{(\texttt{if-exp}\ e_1\ e_2\ e_3)}
\end{aligned}$$

$$(\texttt{proc-exp}\ \mathit{var}\ \mathit{body})\ :\ t_{(\texttt{proc-exp}\ \mathit{var}\ \mathit{body})} = (t_{\mathit{var}} \to t_{\mathit{body}})$$

$$(\texttt{call-exp}\ \mathit{rator}\ \mathit{rand})\ :\ t_{\mathit{rator}} = (t_{\mathit{rand}} \to t_{(\texttt{call-exp}\ \mathit{rator}\ \mathit{rand})})$$

Type Inference: an example



To infer the type of an expression, we'll introduce a type variable for every subexpression and every bound variable, generate the constraints for each subexpression, and then solve the resulting equations. To see how this works, we will infer the types of several sample expressions.

Let us start with the expression `proc (f) proc(x) -((f 3),(f x))`. We begin by making a table of all the bound variables, `proc` expressions, `if` expressions, and procedure calls in this expression, and assigning a type variable to each one.

Expression	**Type Variable**
`f`	t_f
`x`	t_x
`proc(f)proc(x)-((f 3),(f x))`	t_0
`proc(x)-((f 3),(f x))`	t_1
`-((f 3),(f x))`	t_2
`(f 3)`	t_3
`(f x)`	t_4

Type Inference: an example (cont'd)

Now, for each compound expression, we can write down a type equation according to the rules above.

Expression	**Equations**
`proc(f)proc(x)-((f 3),(f x))`	1. $t_0 = t_f \rightarrow t_1$
`proc(x)-((f 3),(f x))`	2. $t_1 = t_x \rightarrow t_2$
`-((f 3),(f x))`	3. $t_3 = \texttt{int}$
	4. $t_4 = \texttt{int}$
	5. $t_2 = \texttt{int}$
`(f 3)`	6. $t_f = \texttt{int} \rightarrow t_3$
`(f x)`	7. $t_f = t_x \rightarrow t_4$

Unification, substitution and bounding

We separate the state of our calculation into the set of equations still to be solved and the substitution found so far. Initially, all of the equations are to be solved, and the substitution found is empty.

Equations	**Substitution**
$t_0 = t_f \rightarrow t_1$	
$t_1 = t_x \rightarrow t_2$	
$t_3 = \texttt{int}$	
$t_4 = \texttt{int}$	
$t_2 = \texttt{int}$	
$t_f = \texttt{int} \rightarrow t_3$	
$t_f = t_x \rightarrow t_4$	

We consider each equation in turn. If the equation's left-hand side is a variable, we move it to the substitution.

Equations	**Substitution**
$t_1 = t_x \rightarrow t_2$	$t_0 = t_f \rightarrow t_1$
$t_3 = \texttt{int}$	
$t_4 = \texttt{int}$	
$t_2 = \texttt{int}$	
$t_f = \texttt{int} \rightarrow t_3$	
$t_f = t_x \rightarrow t_4$	

Type Inference: an example (cont’d)

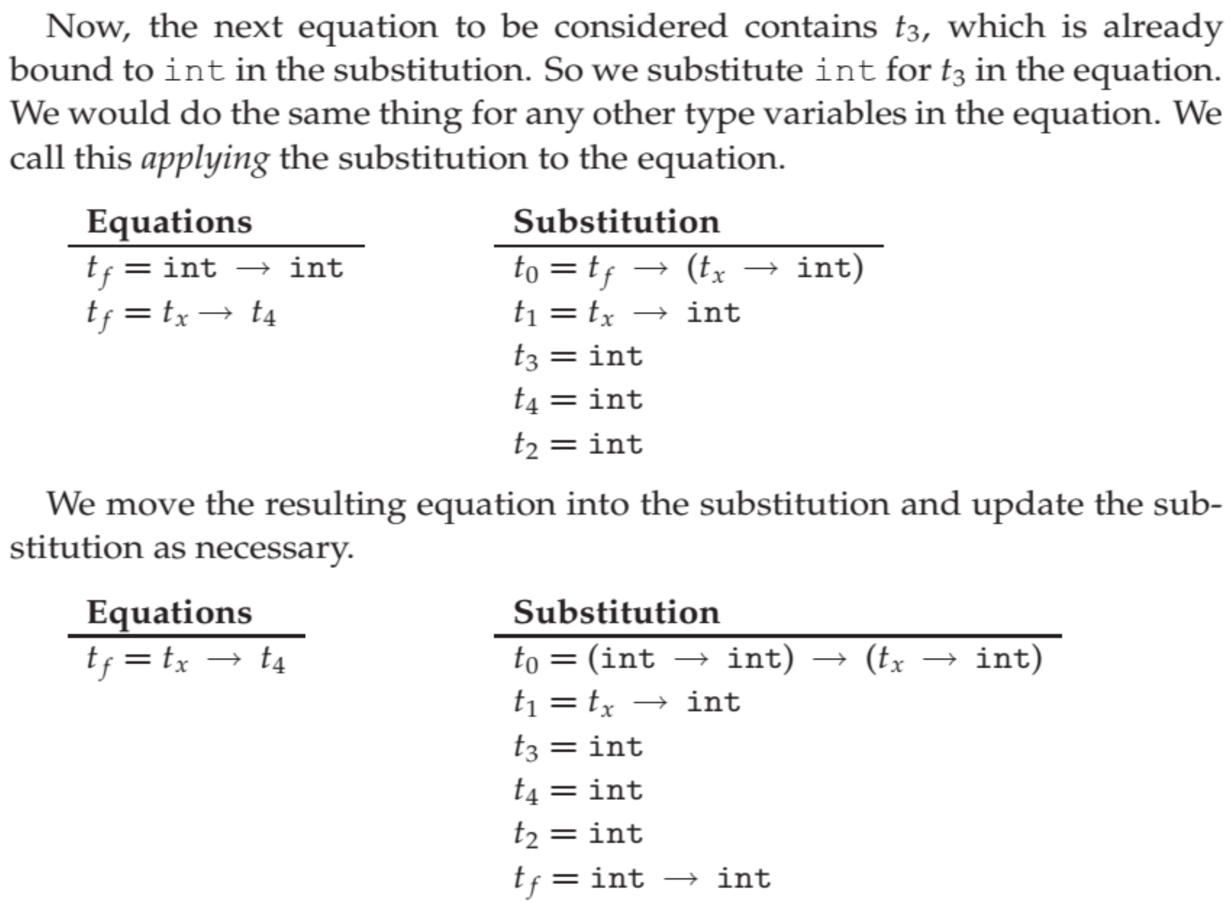
Equations	**Substitution**
$t_3 = \texttt{int}$	$t_0 = t_f \rightarrow (t_x \rightarrow t_2)$
$t_4 = \texttt{int}$	$t_1 = t_x \rightarrow t_2$
$t_2 = \texttt{int}$	
$t_f = \texttt{int} \rightarrow t_3$	
$t_f = t_x \rightarrow t_4$	

Equations	**Substitution**
$t_4 = \texttt{int}$	$t_0 = t_f \rightarrow (t_x \rightarrow t_2)$
$t_2 = \texttt{int}$	$t_1 = t_x \rightarrow t_2$
$t_f = \texttt{int} \rightarrow t_3$	$t_3 = \texttt{int}$
$t_f = t_x \rightarrow t_4$	

Equations	**Substitution**
$t_2 = \texttt{int}$	$t_0 = t_f \rightarrow (t_x \rightarrow t_2)$
$t_f = \texttt{int} \rightarrow t_3$	$t_1 = t_x \rightarrow t_2$
$t_f = t_x \rightarrow t_4$	$t_3 = \texttt{int}$
	$t_4 = \texttt{int}$

Equations	**Substitution**
$t_f = \texttt{int} \rightarrow t_3$	$t_0 = t_f \rightarrow (t_x \rightarrow \texttt{int})$
$t_f = t_x \rightarrow t_4$	$t_1 = t_x \rightarrow \texttt{int}$
	$t_3 = \texttt{int}$
	$t_4 = \texttt{int}$
	$t_2 = \texttt{int}$

Type Inference: an example (cont’d)



Now, the next equation to be considered contains t_3, which is already bound to `int` in the substitution. So we substitute `int` for t_3 in the equation. We would do the same thing for any other type variables in the equation. We call this *applying* the substitution to the equation.

Equations	**Substitution**
$t_f = \texttt{int} \to \texttt{int}$	$t_0 = t_f \to (t_x \to \texttt{int})$
$t_f = t_x \to t_4$	$t_1 = t_x \to \texttt{int}$
	$t_3 = \texttt{int}$
	$t_4 = \texttt{int}$
	$t_2 = \texttt{int}$

We move the resulting equation into the substitution and update the substitution as necessary.

Equations	**Substitution**
$t_f = t_x \to t_4$	$t_0 = (\texttt{int} \to \texttt{int}) \to (t_x \to \texttt{int})$
	$t_1 = t_x \to \texttt{int}$
	$t_3 = \texttt{int}$
	$t_4 = \texttt{int}$
	$t_2 = \texttt{int}$
	$t_f = \texttt{int} \to \texttt{int}$

Type Inference: an example (cont’d)

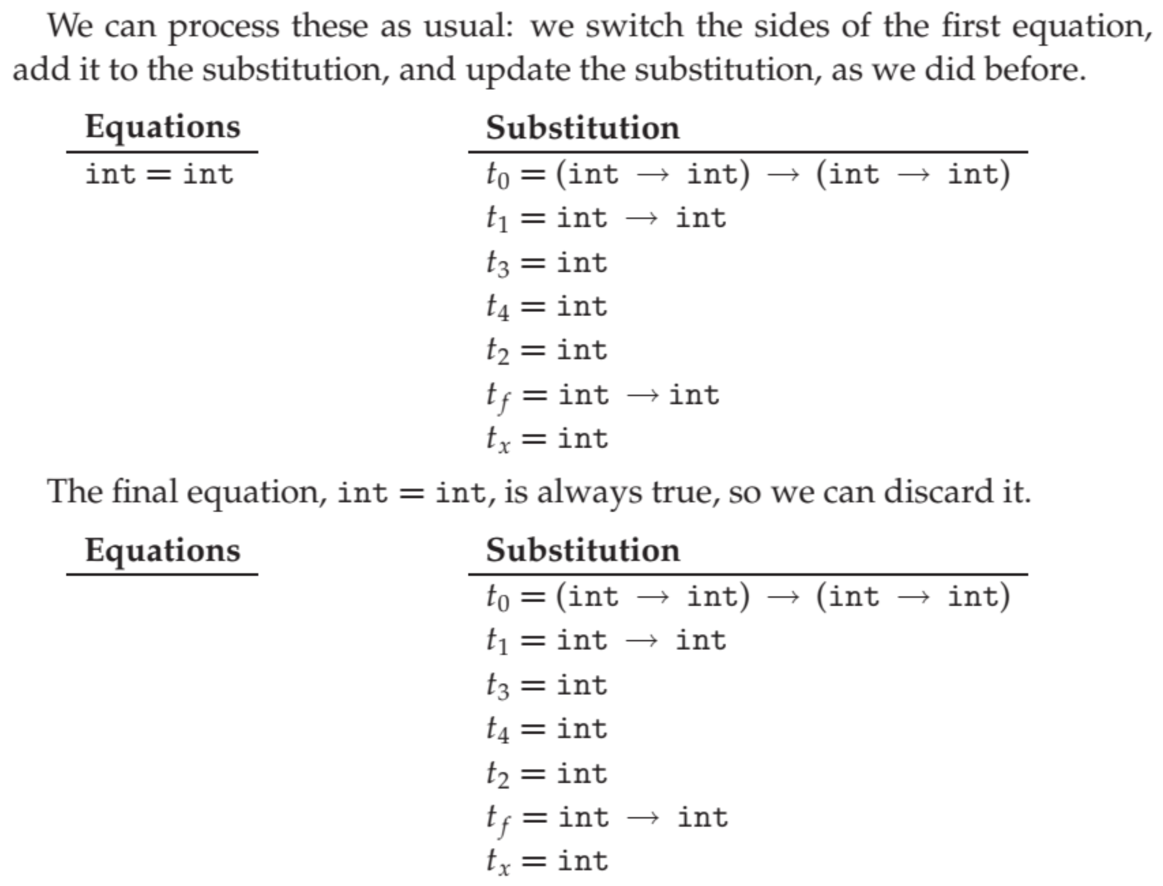
The next equation, $t_f = t_x \rightarrow t_4$, contains t_f and t_4, which are bound in the substitution, so we apply the substitution to this equation. This gets

Equations	Substitution
$\mathtt{int} \rightarrow \mathtt{int} = t_x \rightarrow \mathtt{int}$	$t_0 = (\mathtt{int} \rightarrow \mathtt{int}) \rightarrow (t_x \rightarrow \mathtt{int})$
	$t_1 = t_x \rightarrow \mathtt{int}$
	$t_3 = \mathtt{int}$
	$t_4 = \mathtt{int}$
	$t_2 = \mathtt{int}$
	$t_f = \mathtt{int} \rightarrow \mathtt{int}$

If neither side of the equation is a variable, we can simplify, yielding two new equations.

Equations	Substitution
$\mathtt{int} = t_x$	$t_0 = (\mathtt{int} \rightarrow \mathtt{int}) \rightarrow (t_x \rightarrow \mathtt{int})$
$\mathtt{int} = \mathtt{int}$	$t_1 = t_x \rightarrow \mathtt{int}$
	$t_3 = \mathtt{int}$
	$t_4 = \mathtt{int}$
	$t_2 = \mathtt{int}$
	$t_f = \mathtt{int} \rightarrow \mathtt{int}$

Type Inference: an example (cont'd)

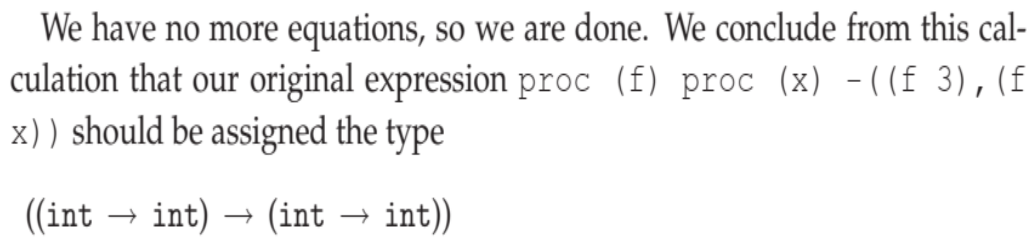


We can process these as usual: we switch the sides of the first equation, add it to the substitution, and update the substitution, as we did before.

Equations	Substitution
$\texttt{int} = \texttt{int}$	$t_0 = (\texttt{int} \to \texttt{int}) \to (\texttt{int} \to \texttt{int})$
	$t_1 = \texttt{int} \to \texttt{int}$
	$t_3 = \texttt{int}$
	$t_4 = \texttt{int}$
	$t_2 = \texttt{int}$
	$t_f = \texttt{int} \to \texttt{int}$
	$t_x = \texttt{int}$

The final equation, $\texttt{int} = \texttt{int}$, is always true, so we can discard it.

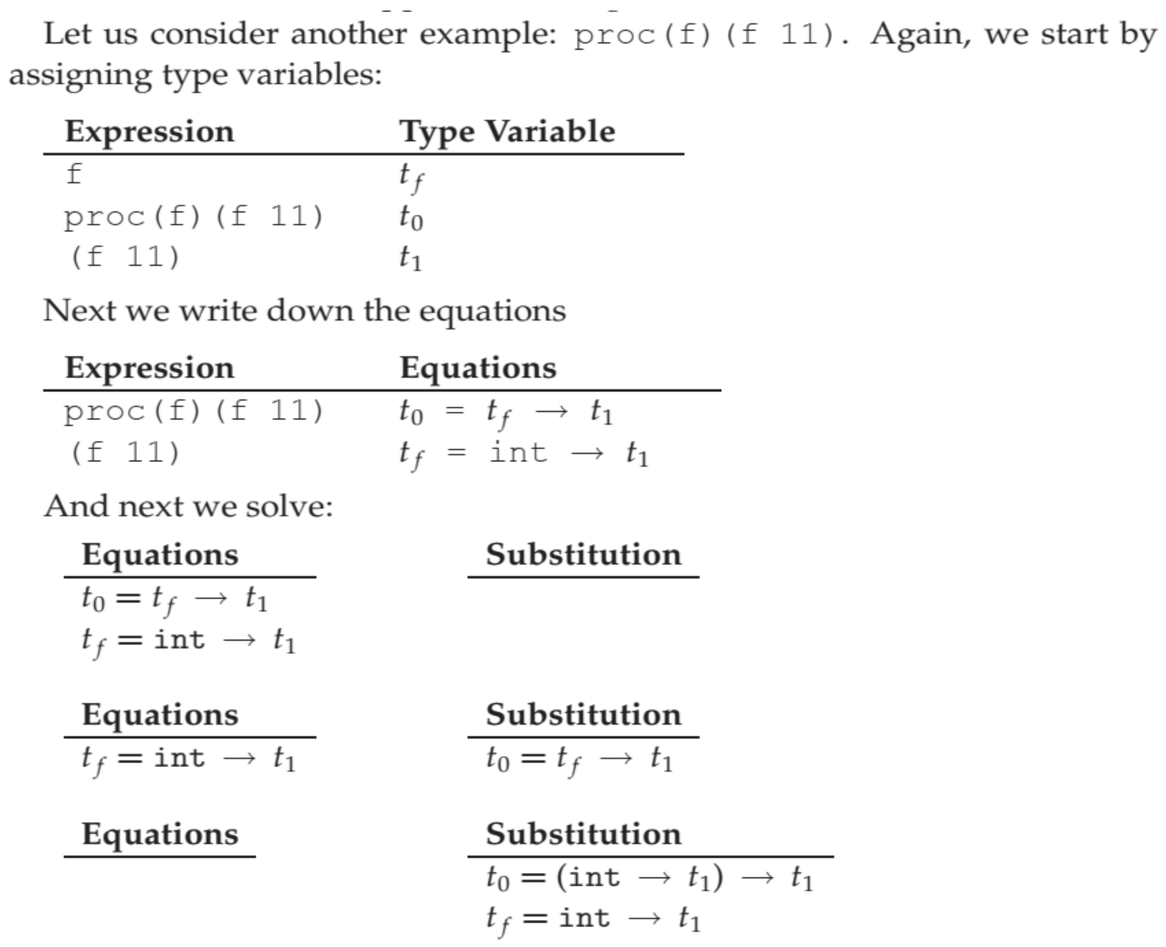
Equations	Substitution
	$t_0 = (\texttt{int} \to \texttt{int}) \to (\texttt{int} \to \texttt{int})$
	$t_1 = \texttt{int} \to \texttt{int}$
	$t_3 = \texttt{int}$
	$t_4 = \texttt{int}$
	$t_2 = \texttt{int}$
	$t_f = \texttt{int} \to \texttt{int}$
	$t_x = \texttt{int}$



We have no more equations, so we are done. We conclude from this calculation that our original expression `proc (f) proc (x) -((f 3),(f x))` should be assigned the type

$((\texttt{int} \to \texttt{int}) \to (\texttt{int} \to \texttt{int}))$

Polymorphic types: an example



Let us consider another example: `proc(f)(f 11)`. Again, we start by assigning type variables:

Expression	Type Variable
`f`	t_f
`proc(f)(f 11)`	t_0
`(f 11)`	t_1

Next we write down the equations

Expression	Equations
`proc(f)(f 11)`	$t_0 = t_f \rightarrow t_1$
`(f 11)`	$t_f = \texttt{int} \rightarrow t_1$

And next we solve:

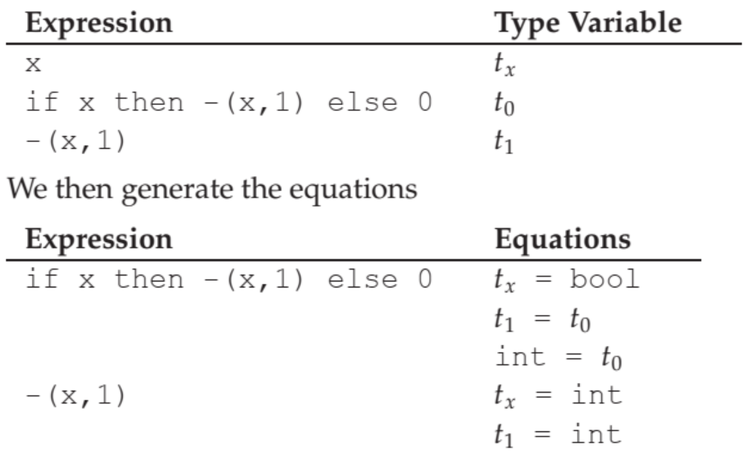
Equations	Substitution
$t_0 = t_f \rightarrow t_1$	
$t_f = \texttt{int} \rightarrow t_1$	

Equations	Substitution
$t_f = \texttt{int} \rightarrow t_1$	$t_0 = t_f \rightarrow t_1$

Equations	Substitution
	$t_0 = (\texttt{int} \rightarrow t_1) \rightarrow t_1$
	$t_f = \texttt{int} \rightarrow t_1$

Type inference: a bad example

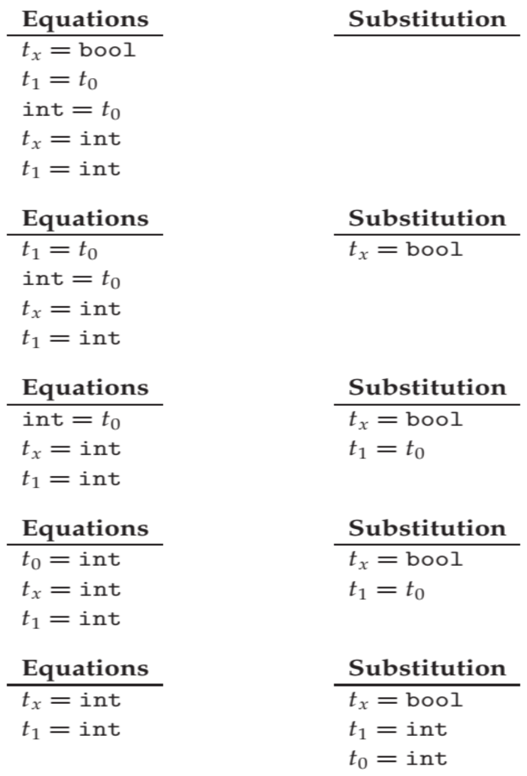
```
if x then -(x,1) else 0
```



Expression	Type Variable
`x`	t_x
`if x then -(x,1) else 0`	t_0
`-(x,1)`	t_1

We then generate the equations

Expression	Equations
`if x then -(x,1) else 0`	$t_x = \texttt{bool}$
	$t_1 = t_0$
	$\texttt{int} = t_0$
`-(x,1)`	$t_x = \texttt{int}$
	$t_1 = \texttt{int}$



Equations	Substitution
$t_x = \texttt{bool}$	
$t_1 = t_0$	
$\texttt{int} = t_0$	
$t_x = \texttt{int}$	
$t_1 = \texttt{int}$	

Equations	Substitution
$t_1 = t_0$	$t_x = \texttt{bool}$
$\texttt{int} = t_0$	
$t_x = \texttt{int}$	
$t_1 = \texttt{int}$	

Equations	Substitution
$\texttt{int} = t_0$	$t_x = \texttt{bool}$
$t_x = \texttt{int}$	$t_1 = t_0$
$t_1 = \texttt{int}$	

Equations	Substitution
$t_0 = \texttt{int}$	$t_x = \texttt{bool}$
$t_x = \texttt{int}$	$t_1 = t_0$
$t_1 = \texttt{int}$	

Equations	Substitution
$t_x = \texttt{int}$	$t_x = \texttt{bool}$
$t_1 = \texttt{int}$	$t_1 = \texttt{int}$
	$t_0 = \texttt{int}$

Equations	Substitution
$\texttt{bool} = \texttt{int}$	$t_x = \texttt{bool}$
$t_1 = \texttt{int}$	$t_1 = \texttt{int}$
	$t_0 = \texttt{int}$

Occurrence Checking

`proc (f) zero?((f f))`

Expression	Type Variable
`proc (f) zero?((f f))`	t_0
`f`	t_f
`zero?((f f))`	t_1
`(f f)`	t_2

Expression	Equations
`proc (f) zero?((f f))`	$t_0 = t_f \rightarrow t_1$
`zero?((f f))`	$t_1 = \texttt{bool}$
	$t_2 = \texttt{int}$
`(f f)`	$t_f = t_f \rightarrow t_2$

Equations	Substitution
$t_f = t_f \rightarrow \texttt{int}$	$t_0 = t_f \rightarrow \texttt{bool}$
	$t_1 = \texttt{bool}$
	$t_2 = \texttt{int}$

So if we ever deduce an equation of the form $tv = t$ where the type variable tv occurs in the type t, we must again conclude that there is no solution to the original equations. This extra condition is called the *occurrence check*.

This condition also means that the substitutions we build will satisfy the following invariant:

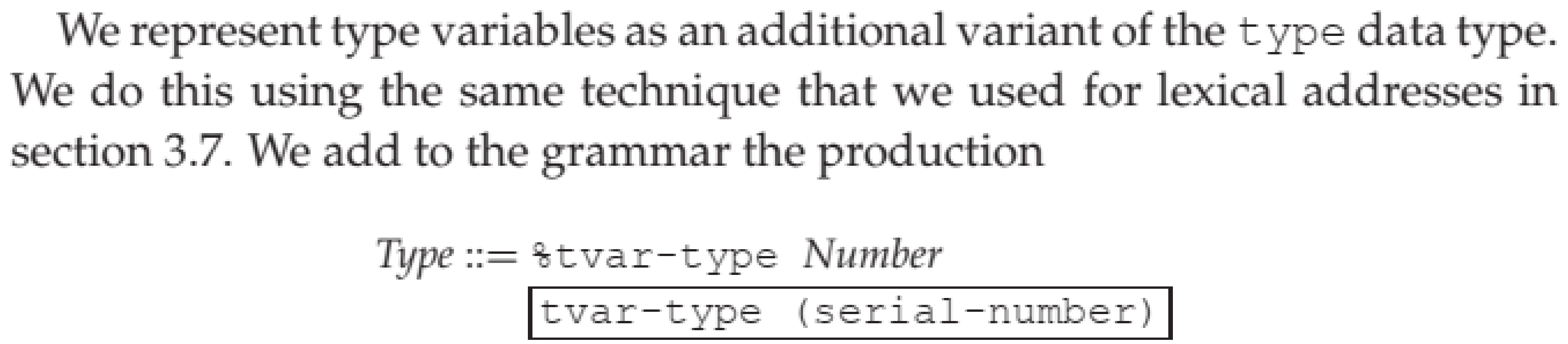
The no-occurrence invariant

No variable bound in the substitution occurs in any of the right-hand sides of the substitution.

Our code for solving equations will depend critically on this invariant.

Type Inference: Implementation

Extending the definition of Type Expressions:

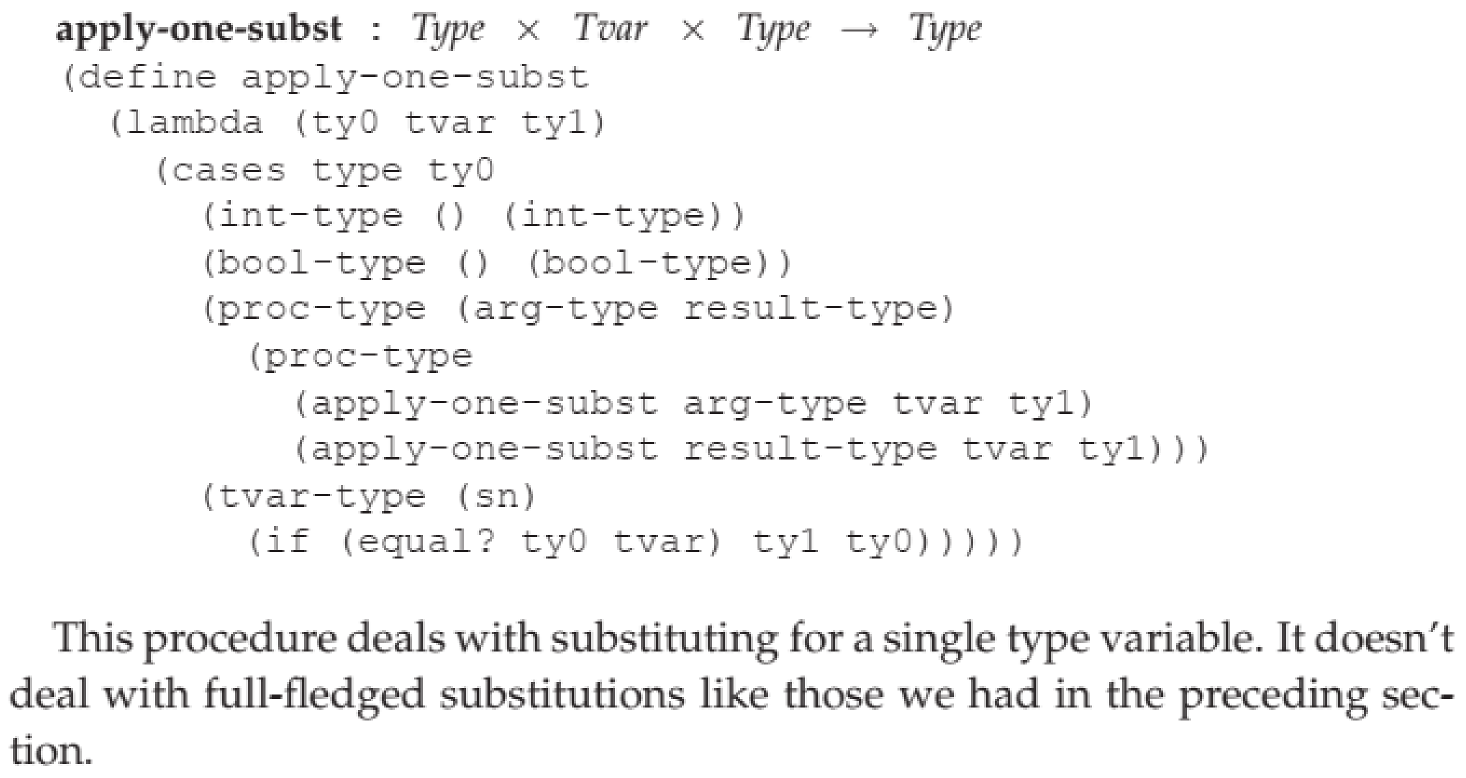


We represent type variables as an additional variant of the `type` data type. We do this using the same technique that we used for lexical addresses in section 3.7. We add to the grammar the production

Type ::= `%tvar-type` *Number*

$\boxed{\texttt{tvar-type (serial-number)}}$

Substitutions: basic operation



apply-one-subst : *Type* × *Tvar* × *Type* → *Type*

```
(define apply-one-subst
  (lambda (ty0 tvar ty1)
    (cases type ty0
      (int-type () (int-type))
      (bool-type () (bool-type))
      (proc-type (arg-type result-type)
        (proc-type
          (apply-one-subst arg-type tvar ty1)
          (apply-one-subst result-type tvar ty1)))
      (tvar-type (sn)
        (if (equal? ty0 tvar) ty1 ty0)))))
```

This procedure deals with substituting for a single type variable. It doesn't deal with full-fledged substitutions like those we had in the preceding section.

Substitutions as a data type (observer)

A substitution is a list of equations between type variables and types. Equivalently, we can think of this list as a function from type variables to types. We say a type variable is *bound* in the substitution if and only if it occurs on the left-hand side of one of the equations in the substitution.

We represent a substitution as a list of pairs (type variable . type). The basic observer for substitutions is apply-subst-to-type. This walks through the type t, replacing each type variable by its binding in the substitution σ. If a variable is not bound in the substitution, then it is left unchanged. We write $t\sigma$ for the resulting type.

The implementation uses the Scheme procedure assoc to look up the type variable in the substitution. assoc returns either the matching (type variable, type) pair or #f if the given type variable is not the car of any pair in the list. We write

apply-subst-to-type : *Type* $\times$ *Subst* $\to$ *Type*

```
(define apply-subst-to-type
  (lambda (ty subst)
    (cases type ty
      (int-type () (int-type))
      (bool-type () (bool-type))
      (proc-type (t1 t2)
        (proc-type
          (apply-subst-to-type t1 subst)
          (apply-subst-to-type t2 subst)))
      (tvar-type (sn)
        (let ((tmp (assoc ty subst)))
          (if tmp
            (cdr tmp)
            ty))))))
```

Substitutions as a data type (constructor)

empty-subst : $() \rightarrow$ *Subst*

```
(define empty-subst (lambda () '()))
```

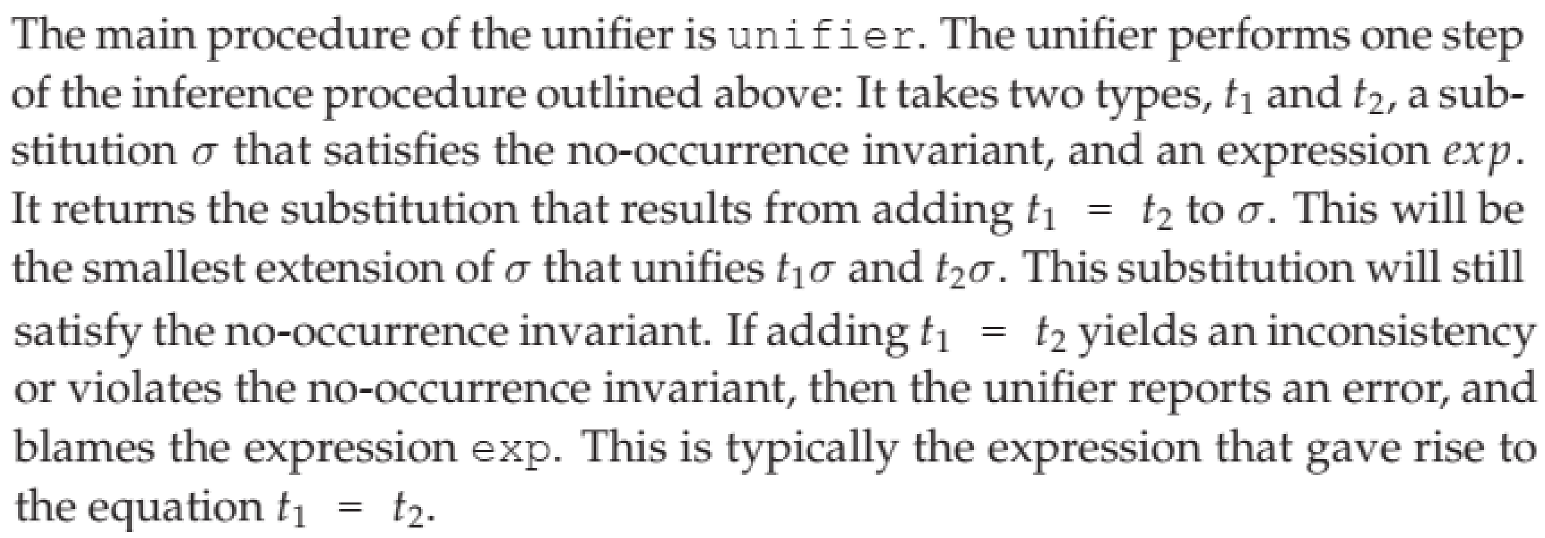
extend-subst : *Subst* $\times$ *Tvar* $\times$ *Type* $\rightarrow$ *Subst*

usage: `tvar not already bound in subst.`

```
(define extend-subst
  (lambda (subst tvar ty)
    (cons
      (cons tvar ty)
      (map
        (lambda (p)
          (let ((oldlhs (car p))
                (oldrhs (cdr p)))
            (cons
              oldlhs
              (apply-one-subst oldrhs tvar ty))))
        subst))))
```

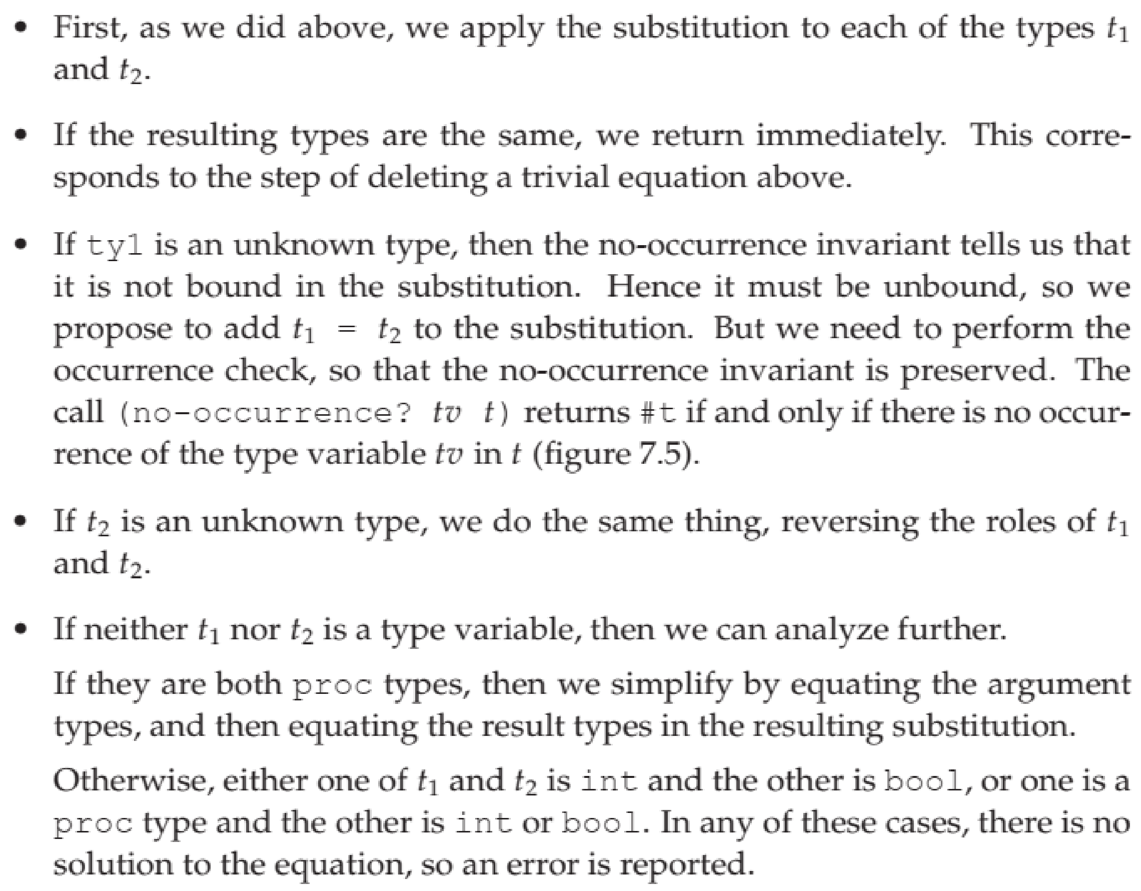
$$\begin{pmatrix} tv_1 = t_1 \\ \vdots \\ tv_n = t_n \end{pmatrix} [tv = t] \quad = \quad \begin{pmatrix} tv \;\; = t \\ tv_1 = t_1[tv = t] \\ \vdots \\ tv_n = t_n[tv = t] \end{pmatrix}$$

Unifier



The main procedure of the unifier is `unifier`. The unifier performs one step of the inference procedure outlined above: It takes two types, t_1 and t_2, a substitution σ that satisfies the no-occurrence invariant, and an expression exp. It returns the substitution that results from adding $t_1 = t_2$ to σ. This will be the smallest extension of σ that unifies $t_1\sigma$ and $t_2\sigma$. This substitution will still satisfy the no-occurrence invariant. If adding $t_1 = t_2$ yields an inconsistency or violates the no-occurrence invariant, then the unifier reports an error, and blames the expression `exp`. This is typically the expression that gave rise to the equation $t_1 = t_2$.

Unifier: algorithm

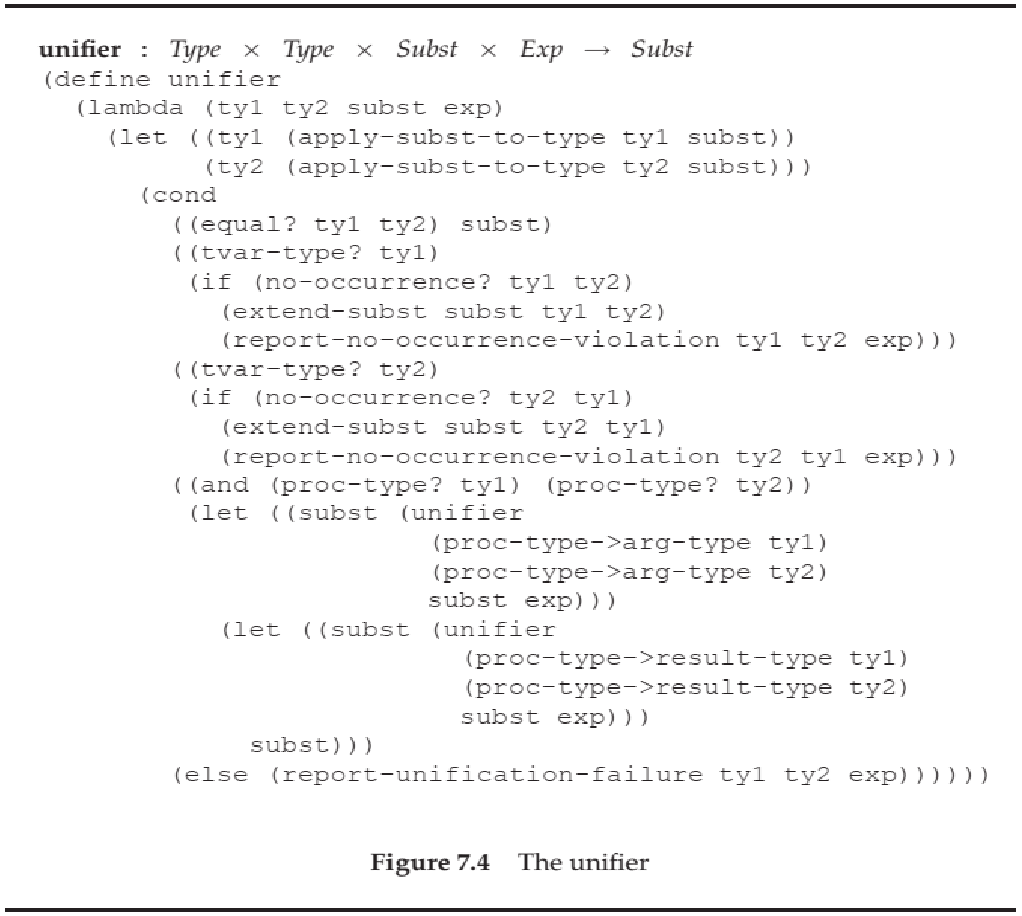


- First, as we did above, we apply the substitution to each of the types t_1 and t_2.
- If the resulting types are the same, we return immediately. This corresponds to the step of deleting a trivial equation above.
- If `ty1` is an unknown type, then the no-occurrence invariant tells us that it is not bound in the substitution. Hence it must be unbound, so we propose to add $t_1 = t_2$ to the substitution. But we need to perform the occurrence check, so that the no-occurrence invariant is preserved. The call `(no-occurrence?` tv t`)` returns `#t` if and only if there is no occurrence of the type variable tv in t (figure 7.5).
- If t_2 is an unknown type, we do the same thing, reversing the roles of t_1 and t_2.
- If neither t_1 nor t_2 is a type variable, then we can analyze further.

  If they are both `proc` types, then we simplify by equating the argument types, and then equating the result types in the resulting substitution.

  Otherwise, either one of t_1 and t_2 is `int` and the other is `bool`, or one is a `proc` type and the other is `int` or `bool`. In any of these cases, there is no solution to the equation, so an error is reported.

Unifier: code

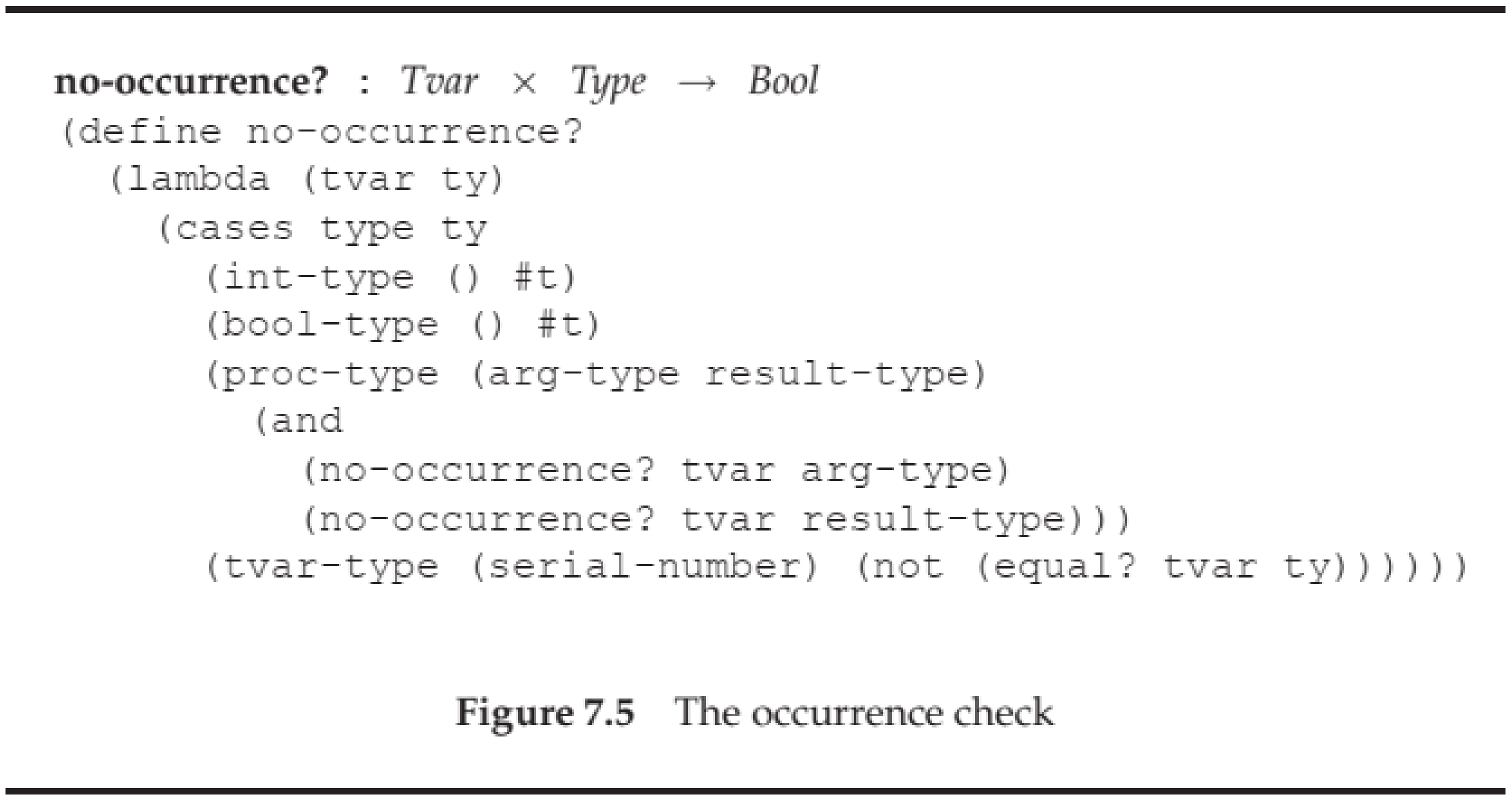


unifier : *Type* × *Type* × *Subst* × *Exp* → *Subst*

```
(define unifier
  (lambda (ty1 ty2 subst exp)
    (let ((ty1 (apply-subst-to-type ty1 subst))
          (ty2 (apply-subst-to-type ty2 subst)))
      (cond
        ((equal? ty1 ty2) subst)
        ((tvar-type? ty1)
         (if (no-occurrence? ty1 ty2)
           (extend-subst subst ty1 ty2)
           (report-no-occurrence-violation ty1 ty2 exp)))
        ((tvar-type? ty2)
         (if (no-occurrence? ty2 ty1)
           (extend-subst subst ty2 ty1)
           (report-no-occurrence-violation ty2 ty1 exp)))
        ((and (proc-type? ty1) (proc-type? ty2))
         (let ((subst (unifier
                        (proc-type->arg-type ty1)
                        (proc-type->arg-type ty2)
                        subst exp)))
           (let ((subst (unifier
                          (proc-type->result-type ty1)
                          (proc-type->result-type ty2)
                          subst exp)))
             subst)))
        (else (report-unification-failure ty1 ty2 exp))))))
```

Figure 7.4 The unifier

No-Occurrence Checking



no-occurrence? : *Tvar* × *Type* → *Bool*

```
(define no-occurrence?
  (lambda (tvar ty)
    (cases type ty
      (int-type () #t)
      (bool-type () #t)
      (proc-type (arg-type result-type)
        (and
          (no-occurrence? tvar arg-type)
          (no-occurrence? tvar result-type)))
      (tvar-type (serial-number) (not (equal? tvar ty))))))
```

Figure 7.5 The occurrence check

Finding type of an expression

optype->type : *OptionalType* → *Type*

```
(define otype->type
  (lambda (otype)
    (cases optional-type otype
      (no-type () (fresh-tvar-type))
      (a-type (ty) ty))))
```

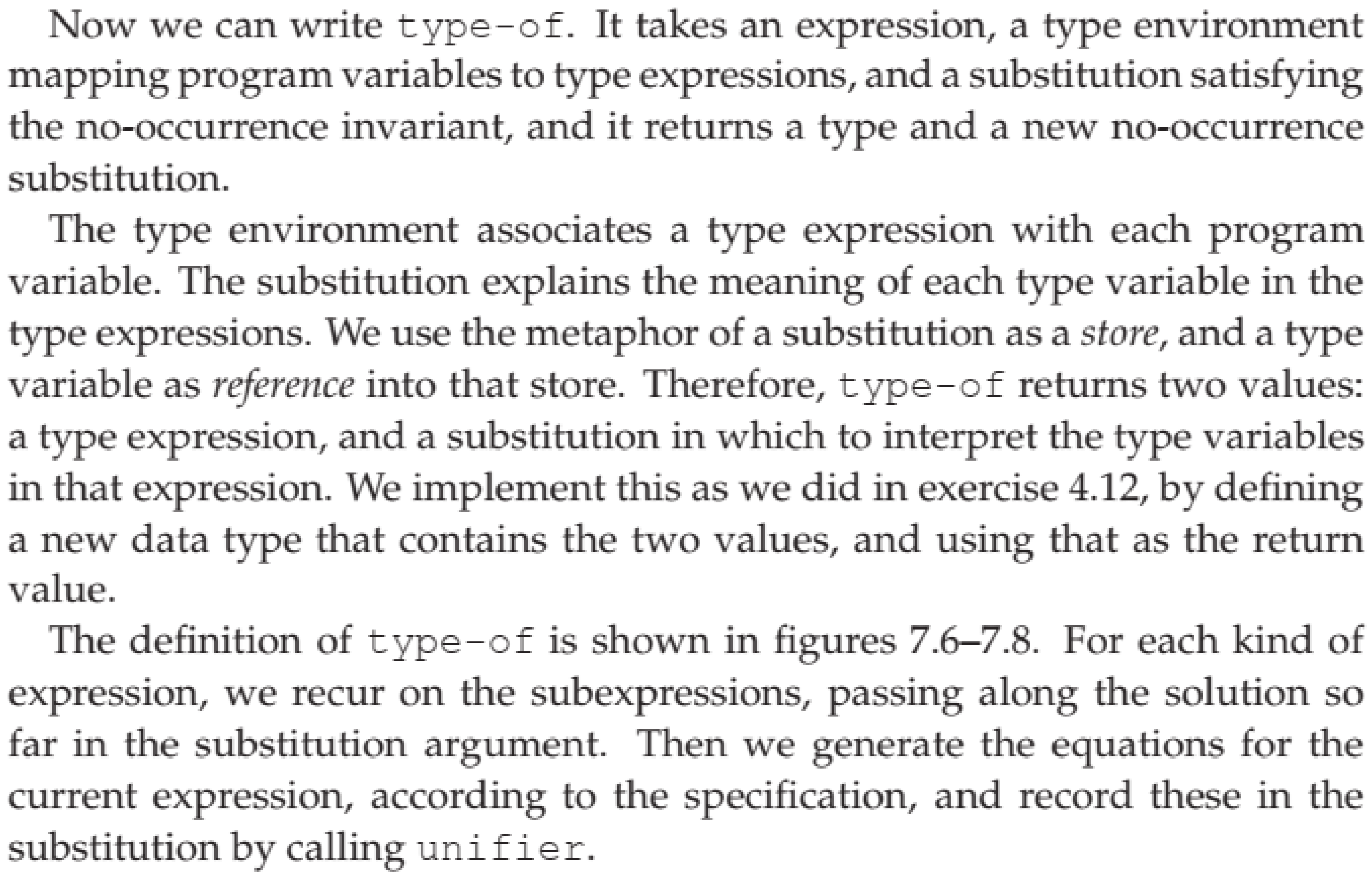
fresh-tvar-type : () → *Type*

```
(define fresh-tvar-type
  (let ((sn 0))
    (lambda ()
      (set! sn (+ sn 1))
      (tvar-type sn))))
```

type-to-external-form : *Type* → *List*

```
(define type-to-external-form
  (lambda (ty)
    (cases type ty
      (int-type () 'int)
      (bool-type () 'bool)
      (proc-type (arg-type result-type)
        (list
          (type-to-external-form arg-type)
          '->
          (type-to-external-form result-type)))
      (tvar-type (serial-number)
        (string->symbol
          (string-append
            "ty"
            (number->string serial-number)))))))
```

Type-of



Now we can write `type-of`. It takes an expression, a type environment mapping program variables to type expressions, and a substitution satisfying the no-occurrence invariant, and it returns a type and a new no-occurrence substitution.

The type environment associates a type expression with each program variable. The substitution explains the meaning of each type variable in the type expressions. We use the metaphor of a substitution as a *store*, and a type variable as *reference* into that store. Therefore, `type-of` returns two values: a type expression, and a substitution in which to interpret the type variables in that expression. We implement this as we did in exercise 4.12, by defining a new data type that contains the two values, and using that as the return value.

The definition of `type-of` is shown in figures 7.6–7.8. For each kind of expression, we recur on the subexpressions, passing along the solution so far in the substitution argument. Then we generate the equations for the current expression, according to the specification, and record these in the substitution by calling `unifier`.

Answer = *Type* × *Subst*

```
(define-datatype answer answer?
  (an-answer
    (ty type?)
    (subst substitution?)))
```

type-of-program : *Program* → *Type*

```
(define type-of-program
  (lambda (pgm)
    (cases program pgm
      (a-program (exp1)
        (cases answer (type-of exp1
                        (init-tenv) (empty-subst))
          (an-answer (ty subst)
            (apply-subst-to-type ty subst)))))))
```

type-of : *Exp* × *Tenv* × *Subst* → *Answer*

```
(define type-of
  (lambda (exp tenv subst)
    (cases expression exp

      (const-exp (num) (an-answer (int-type) subst))
```

(zero?-exp e_1) : $t_{e_1} = \textbf{int}$

$t_{(\texttt{zero?-exp}\ e_1)} = \textbf{bool}$

```
      (zero?-exp (exp1)
        (cases answer (type-of exp1 tenv subst)
          (an-answer (ty1 subst1)
            (let ((subst2
                    (unifier ty1 (int-type) subst1 exp)))
              (an-answer (bool-type) subst2)))))
```

(diff-exp e_1 e_2) : $t_{e_1} = \textbf{int}$

$t_{e_2} = \textbf{int}$

$t_{(\texttt{diff-exp}\ e_1\ e_2)} = \textbf{int}$

```
(diff-exp (exp1 exp2)
  (cases answer (type-of exp1 tenv subst)
    (an-answer (ty1 subst1)
      (let ((subst1
              (unifier ty1 (int-type) subst1 exp1)))
        (cases answer (type-of exp2 tenv subst1)
          (an-answer (ty2 subst2)
            (let ((subst2
                    (unifier ty2 (int-type)
                      subst2 exp2)))
              (an-answer (int-type) subst2)))))))
```

(if-exp e_1 e_2 e_3) : $t_{e_1} = \textbf{bool}$

$t_{e_2} = t_{(\texttt{if-exp}\ e_1\ e_2\ e_3)}$

$t_{e_3} = t_{(\texttt{if-exp}\ e_1\ e_2\ e_3)}$

```
(if-exp (exp1 exp2 exp3)
  (cases answer (type-of exp1 tenv subst)
    (an-answer (ty1 subst)
      (let ((subst
              (unifier ty1 (bool-type) subst exp1)))
        (cases answer (type-of exp2 tenv subst)
          (an-answer (ty2 subst)
            (cases answer (type-of exp3 tenv subst)
              (an-answer (ty3 subst)
                (let ((subst
                        (unifier ty2 ty3 subst exp)))
                  (an-answer ty2 subst)))))))))

(var-exp (var)
  (an-answer (apply-tenv tenv var) subst))

(let-exp (var exp1 body)
  (cases answer (type-of exp1 tenv subst)
    (an-answer (exp1-type subst)
      (type-of body
        (extend-tenv var exp1-type tenv)
        subst))))
```

$$\boxed{\texttt{(proc-exp}\ \mathit{var}\ \mathit{body}\texttt{)} \quad : \quad t_{\texttt{(proc-exp}\ \mathit{var}\ \mathit{body}\texttt{)}} = (t_{var} \ \rightarrow \ t_{body})}$$

```
(proc-exp (var otype body)
  (let ((var-type (otype->type otype)))
    (cases answer (type-of body
                    (extend-tenv var var-type tenv)
                    subst)
      (an-answer (body-type subst)
        (an-answer
          (proc-type var-type body-type)
          subst)))))
```

$$\boxed{\texttt{(call-exp}\ \mathit{rator}\ \mathit{rand}\texttt{)} \quad : \quad t_{rator} = (t_{rand} \ \rightarrow \ t_{\texttt{(call-exp}\ \mathit{rator}\ \mathit{rand}\texttt{)}})}$$

```
(call-exp (rator rand)
  (let ((result-type (fresh-tvar-type)))
    (cases answer (type-of rator tenv subst)
      (an-answer (rator-type subst)
        (cases answer (type-of rand tenv subst)
          (an-answer (rand-type subst)
            (let ((subst
                    (unifier
                      rator-type
                      (proc-type
                        rand-type result-type)
                      subst
                      exp)))
              (an-answer result-type subst))))))))
```

$$\boxed{\begin{array}{l} \texttt{letrec}\ t_{proc-result}\ p\ (var\ :\ t_{var}) = e_{proc\text{-}body}\ \texttt{in}\ e_{letrec\text{-}body} \quad : \\ \quad t_p = t_{var} \rightarrow t_{e_{proc\text{-}body}} \\ \quad t_{e_{letrec\text{-}body}} = t_{\texttt{letrec}\ t_{proc-result}\ p\ (var\ :\ t_{var}) = e_{proc\text{-}body}\ \texttt{in}\ e_{letrec\text{-}body}} \end{array}}$$

```
(letrec-exp (p-result-otype p-name b-var b-var-otype
             p-body letrec-body)
  (let ((p-result-type (otype->type p-result-otype))
        (p-var-type (otype->type b-var-otype)))
    (let ((tenv-for-letrec-body
            (extend-tenv p-name
              (proc-type p-var-type p-result-type)
             tenv)))
      (cases answer (type-of p-body
                      (extend-tenv b-var p-var-type
                        tenv-for-letrec-body)
                      subst)
        (an-answer (p-body-type subst)
          (let ((subst
                  (unifier p-body-type p-result-type
                    subst p-body)))
            (type-of letrec-body
             tenv-for-letrec-body
             subst)))))))
```